Dottorato di Ricerca in Informatica
Università di Bologna e Padova

Interactive Theorem Provers: issues faced as a user and tackled as a developer

Enrico Tassi

March 2008

Coordinatore:	Tutore:
Ozalp Babaöglu	Andrea Asperti

Abstract

Interactive theorem provers (ITP for short) are tools whose final aim is to certify proofs written by human beings. To reach that objective they have to fill the gap between the high level language used by humans for communicating and reasoning about mathematics and the lower level language that a machine is able to "understand" and process. The user perceives this gap in terms of missing features or inefficiencies. The developer tries to accommodate the user requests without increasing the already high complexity of these applications. We believe that satisfactory solutions can only come from a strong synergy between users and developers.

We devoted most part of our PHD designing and developing the Matita interactive theorem prover. The software was born in the computer science department of the University of Bologna as the result of composing together all the technologies developed by the HELM team (to which we belong) for the MoWGLI project. The MoWGLI project aimed at giving accessibility through the web to the libraries of formalised mathematics of various interactive theorem provers, taking Coq as the main test case. The motivations for giving life to a new ITP are:

- study the architecture of these tools, with the aim of understanding the source of their complexity
- exploit such a knowledge to experiment new solutions that, for backward compatibility reasons, would be hard (if not impossible) to test on a widely used system like Coq.

MATITA is based on the Curry-Howard isomorphism, adopting the Calculus of Inductive Constructions (CIC) as its logical foundation. Proof objects are thus, at some extent, compatible with the ones produced with the Coq ITP, that is itself able to import and process the ones generated using MATITA. Although the systems have a lot in common, they share no code at all, and even most of the algorithmic solutions are different.

The thesis is composed of two parts where we respectively describe our experience as a user and a developer of interactive provers. In particular, the first part is based on two different formalisation experiences:

- our internship in the Mathematical Components team (INRIA), that is formalising the finite group theory required to attack the Feit Thompson Theorem. To tackle this result, giving an effective classification of finite groups of odd order, the team adopts the SSREFLECT Coq extension, developed by Georges Gonthier for the proof of the four colours theorem.

- our collaboration at the D.A.M.A. Project, whose goal is the formalisation of abstract measure theory in MATITA leading to a constructive proof of Lebesgue's Dominated Convergence Theorem.

The most notable issues we faced, analysed in this part of the thesis, are the following: the difficulties arising when using "black box" automation in large formalisations; the impossibility for a user (especially a newcomer) to master the context of a library of already formalised results; the uncomfortable big step execution of proof commands historically adopted in ITPs; the difficult encoding of mathematical structures with a notion of inheritance in a type theory without subtyping like CIC.

In the second part of the manuscript many of these issues will be analysed with the looking glasses of an ITP developer, describing the solutions we adopted in the implementation of MATITA to solve these problems: integrated searching facilities to

assist the user in handling large libraries of formalised results; a small step execution semantic for proof commands; a flexible implementation of coercive subtyping allowing multiple inheritance with shared substructures; automatic tactics, integrated with the searching facilities, that generates proof commands (and not only proof objects, usually kept hidden to the user) one of which specifically designed to be user driven.

Acknowledgements

The research team leaded by Professor Asperti has always been warm with me, allowing me to work in a friendly and productive atmosphere from the very beginning. The first thanks goes to the team as a whole.

My family never stopped giving me psychological and economical support during these three years, thank you very much. My best research result, also known as Cinzia, filled my life with so many attentions and sincere feelings I think nobody could ask for more. Thank you sweetie.

My advisor Professor Andrea Asperti never stopped involving me in research activities, working with me side by side as much as possible. I wish the same amount of collaboration to every PHD student.

I'm grateful to Georges Gonthier for giving me the opportunity to work abroad in his team in Orsay (Paris) for six months, spending many hours working with me side by side. I wish the same good experience to every PHD student.

Stefano Zacchiroli has been my Debian advocate almost three years ago: I've never expected so much fun coming from my work on such project. No coffee pause would have been that fun without chatting with him about the latest Debian-related news.

Claudio Sacerdoti's help as always been invaluable during my research activity. I still try to understand how he always finds a good example to explain what he has in mind.

A special thanks goes to Assia and Ricardo, that during my six months internship have been more then just colleagues. Another special thank goes to Martine whose help has been invaluable in many occasions.

Last, but not least, a thank goes to my “internal” referees, Professor Sangiorgi and Professor Gabbrielli for giving me, maybe unconsciously, injections of self esteem during every annual report.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We devoted most part of our PHD designing and developing the MATITA [3] interactive theorem prover. The software was born in the computer science department of the University of Bologna as the result of composing together all the technologies developed by the HELM[1] team (to which we belong) for the MoWGLI[2] project.

The MoWGLI project aimed at giving accessibility through the web to the libraries of formalised mathematics of various interactive theorem provers, taking Coq [92] as the main test case. The motivations for giving life to a new ITP are:

- study the architecture of these tools, with the aim of understanding the source of their complexity
- exploit such a knowledge to experiment new solutions that, for backward compatibility reasons, would be hard (if not impossible) to test on a widely used system like Coq.

These motivations come from a simple observation. Automatic theorem provers showed their effectiveness many times [35, 86], and their architecture and design choices are really well detailed in the literature [56, 78, 76]. Moreover, comparisons between different solutions adopted are performed in a pragmatic way [67, 60, 76, 77] giving a clean classification of algorithms and data structures, analysing their tips and pitfalls. Said that, tools like the well known Otter [63] count around 50 thousands lines of C code, the impressive Vampire [78] a little more then 100 thousands of C++, the same size for Waldmeister [19]. Coq is written in Objective Caml, that is notably less verbose than C or C++, but counts more than 110 thousands lines of code (version 8). It works in higher order logic, and not the first order one implemented in most automatic theorem provers, but its objective, proof checking, is usually considered a simpler, less ambitious, task than proof search. Moreover, the size of the kernel of Coq is 10 thousands lines of code and it is hard to believe that to build up a successful interactive theorem prover one has to write

[1]Hypertextual Electronic Library of Mathematics

[2]MoWGLI: Mathematics On the Web: Get it by Logic Interfaces. IST-2001-33562.

an additional 100 thousands lines of code. The architecture of interactive theorem provers is not pragmatically studied as it is done for automatic theorem provers. The HELM team decided to investigate the source of the complexity of Coq trying to rewrite it from scratch, possibly documenting tips an pitfalls learned in this process and comparing the solutions adopted with the ones already implemented in Coq.

Matita is based on the Curry-Howard isomorphism, adopting the Calculus of Inductive Constructions [93, 73, 30] (CIC) as its logical foundation. Proof objects are thus, at some extent, compatible with the ones produced with the Coq ITP, that is itself able to import and process the ones generated using Matita. Although the systems have a lot in common, they share no code at all, and even most of the algorithmic solutions are different.

The thesis is composed of two parts where we respectively describe our experience as a user and a developer of interactive provers, pointing out issues we faced as a user that are inherent to the subsystems of interactive theorem provers we worked on as a developer.

The first part of the manuscript is based on two different formalisation experiences, using the interactive theorem prover Coq in the first and Matita in the second.

The former experience comes from our internship in the Mathematical Components team[3] (INRIA) leaded by Gonthier. The Mathematical Components project aims to demonstrate that formalised mathematical theories can, like modern software, be built out of components. To reach this objective, it proposes to develop a general platform for mathematical components, based on the Coq SSReflect [45] extension, developed by Gonthier to carry out the formalisation of the Four Colour Theorem [44]. The team is formalising the finite group theory required to attack the Feit Thompson Theorem and we worked at the formalisation of many components on which the whole formalisation will stand, like finite sets. The results we formalised as well as the issues we encountered are detailed in Chapter 2. Our con-

[3]http://www.msr-inria.inria.fr/Projects/math-components/

tribution has been published in [46] that we co-authored with other members of the Mathematical Components team: Gonthier, Mahboubi, Rideau and Thery.

The latter experience comes from our collaboration at the D.A.M.A. Project[4], leaded by Sacerdoti Coen and funded by the University of Bologna. The projects has three major goals:

- improvement and specialisation of the interactive theorem prover MATITA
- development on top of MATITA of a learning environment for students to verify their improvements in doing mathematical proofs
- formalisation of abstract measure theory in MATITA up to Lebesgue's Dominated Convergence Theorem

We worked on the first and the last items, but in the first part of the thesis we concentrate only on the last task. Our work consisted in the encoding of algebraic structures and abstract measure notions in constructive type theory. We then proved the so called sandwich lemma in this constructive setting.

One of the most notable issues we faced as a user, analysed in this part of the thesis, is the difficulty for a newcomer (like we were at the beginning of our experience in the Mathematical Components team) to master the context of a reasonably large library of available results. The library in question was mainly developed for the four colour theorem and is reasonably large and well designed (with standard naming schemas for example). Although Coq offers some searching facilities, we found them insufficient. Gonthier himself later developed a more sophisticated "Search" command to ameliorate the situation. We also believe that ITPs can not be conceived thinking that a (trained) user masters the content of the available library, since the effort needed to formalise mathematical results is known to be non negligible and is thus realistic to think that entire teams collaborate on the same formalisation. It is hardly the case that every team component knows exactly what

[4]`http://dama.cs.unibo.it/`

all her colleagues did. Moreover, available theorems are usually referenced by name (or even sequential numbers in the huge library of Mizar [64]) and the user needs to remember the precise statement and recall the exact name to use it.

Another issue is that the tactic engine of Coq executes composed tactic in a big step fashion. SSReflect script lines are the composition of many simpler tactics and, even if the syntax allows to compose them in an easy way, the big step execution model of Coq makes their incremental construction particularly tedious. The user is forced to undo a command to compose it with the following one.

During our internship in the Mathematic Components team we never used automatic tactics. The main motivation was that proof scripts that heavily uses automatic tactics are harder to mend when they break. Finding the right and most handy definition is an hard task, and since there is no silver bullet, they are always subject to changes. These changes are usually made to make proofs work smoother, and break many existing already proved results that have thus to be fixed, possibly shortening the proof because of the new smarter definition. This process is long and tedious, but it is still feasible and contributes in keeping the library of formalised results in a good shape. When a proof heavily use automation and breaks, the user has usually no idea of what went wrong, since no trace of the previously found proof is left to him. Moreover, even if automation does not break, the lack of an history of what was done in the previous run forces the tactic to find again a proof, eventually the same. This makes the execution of proof scripts slower and the user tends to procrastinate tasks like moving lemmas in the right place just because she does not want to trigger the re-execution of many proof scripts. That last issue is not only related to automation, but to the linear execution paradigm proposed by Proof General [5].

The last issue that will be treated in this thesis is the well known difficulty to encode in a type theory without subtyping, like CIC, mathematical structures that respect an inheritance relation. That problem is usually described in terms of coercive subtyping [57] and dependently typed records [31, 15] that, together, allow to mimic inheritance. Coercive subtyping has been widely implemented in ITPs, but strong restrictions are usually made to ease the implementation, constraining the user to encode mathematical structures in a non natural way.

In the second part of the manuscript many of these issues will be analysed with the looking glasses of an ITP developer, describing the solutions we adopted in the implementation of Matita to attack these problems. Although we worked on almost every component of Matita we describe in full details the refiner (type inference) subsystem in Chapter 5 and the automatic tactics in Chapter 6.

A preliminary introduction to Matita is made in Chapter 4 giving a data-driven analysis of overall architecture of the tool. Particular attention is given to some peculiarities of the system we consider relevant, like the integrated search engine Whelp and the small-step execution tactic language. The chapter concludes describing the solutions we adopted to deliver a complex system like Matita to the users. Matita always suffered from complex installation procedures and external dependencies, like a relational database, making it almost impossible to concretely release it to the public. The system has now an official release, can be evaluated without even installing it by means of a live CD and can be installed with a single click on Linux distribution based on Debian[5] GNU/Linux, like the nowadays widespread Ubuntu[6]. Our contributions described in this chapter are related to Whelp search engine we tuned for performances and are published in [1] we co-author with the rest of the HELM team, and the small step tactic language [81] that we designed and developed together with Sacerdoti and Zacchiroli. The part regarding the delivery of the system is our contribution too.

Chapter 5 describes the implementation of coercive subtyping we made in Matita and the way it has been used to formalise algebraic structures for the DAMA Project.

Most implementation of coercive subtyping, like the ones of Coq and Lego, do not allow multiple coercive paths between the same types. Allowing that, the user can

[5]`http://debian.org`
[6]`http://ubuntu.com`

build algebraic structures in a more natural way, possibly creating diamonds. For example, let us call a ring a structure with a carrier and two operations, the former forming with the carrier a group and the latter a monoid, and declare coercions from ring to monoid (called ψ_2) and group (called ϕ_2). Both group and monoid are then coercible to their carrier, a type, by ϕ_1 and ψ_1.

$$\begin{tikzcd}
 & \boxed{Ring} \arrow[dl, "\phi_2"'] \arrow[dr, "\psi_2"] & \\
\boxed{Group} \arrow[dr, "\phi_1"'] & & \boxed{Monoid} \arrow[dl, "\psi_1"] \\
 & \boxed{Type} &
\end{tikzcd}$$

In this scenario two distinct paths from a ring to the (same) carrier are available, and uncommon unification problem arise in formulas using both the monoid and the group operations (respectively $+$ and $*$), whose input/output type has to be unified. In a formula like $x + (y * z)$ we find a rigid v.s. rigid case, where the output of $*$ has to be unified with the input of $+$. In that case both head constants are coercions and the arguments (the structure the operation is projecting) are flexible: $\psi_1\ ?_1 \overset{?}{\equiv} \phi_1\ ?_2$. ϕ_1 and ψ_1 are different constants, thus unification heuristics usually fail in cases like this one. But this unification problem has a solution that can be found exploiting the information given by the inheritance graph: there is a structure (the ring) containing a group and monoid with the same carrier, thus the former unification problem can be reduced to the following one:

$$\psi_1\ (\psi_2\ ?_3) \overset{?}{\equiv} \phi_1\ (\phi_2\ ?_3)$$

The typing rules adopted in the refiner subsystem are presented, as well as the modification made to the original unification algorithm of MATITA.

Additionally, subset [87] coercions can be declared. Subset coercions, when applied, generate side proofs asserting that the coerced element validates a given predicate. An example of such coercion is the one mapping a list l to the sigma type of ordered lists. When this coercion is applied to a list, the conjecture that such list is ordered is opened, and the user is asked to prove it. Propagating subset coercions under fix points and pattern matching constructors gives a mechanism, in the spirit of PVS predicate subtyping [85], to specify software.

Our contribution has been published with Sacerdoti in [27] and details the encoding of mathematical structures in CIC we made as well as the unification algorithm that exploits the inheritance relation expressed in the graph of coercions.

Chapter 6 is devoted to the description of two automatic tactics we implemented and how we integrated them in an interactive tool like Matita. Both tactics generate proof scripts starting from carefully built proof objects.

We believe that generating proof scripts allows to drop the usual black box nature of automation, possibly allowing the user to adopt automated tactics more fruitfully even in frequently changing developments. Re-execution of proof script is obviously faster since the proof is given in the script file and failures, due to a modified definition for example, can be clearly spotted since the previously found proof breaks when executing a precise command (i.e. the application of a lemma). Black box automation can lead to extremely compact proof scripts, and in some cases make minor modifications to definitions transparent. We believe that compactness of proof script can mainly be regarded as an editor issue, proofs found by means of automation could be folded if too long: any modern editor supports folding. Moreover, the ability of automation to make proof scripts resistant to small changes is still available, the generated proof script can, in case of failure, execute the automatic tactic again, possibly replacing the previously generated proof script with a new one.

Another requirement we made on automation is to be integrated with searching facilities. Huge libraries are likely to be created by many ITP users: one can not assume that every user knows the whole content of the library. Providing a set of lemmas to an automatic tactic can be seen as an hint, speeding up the searching procedure cutting down the search space, not as the only way a tactic works. Already available results concerning the current goal have to be suggested by the system to the user, that may be not aware of their existence.

We implemented two tactics, both generating proof scripts starting from carefully refined proof objects and exploiting the searching facilities available in Matita. One performs (first order) rewriting using the superposition calculus, and is optimised for speed obtaining good results against the enormous TPTP [90] test suite. The other tactic was designed having user interaction in mind. It performs Prolog-like proof search using a depth-first strategy. This, at the cost of worse performances, allows a good user interaction that is thus able to prune/follow computations. Our contribution to the proof reconstruction algorithm has been published in [4] we co-author with our advisor Professor Asperti. The implementation and tuning of both automatic tactics is also our contribution.

Part I

A user perspective

Chapter 2

Mathematical components

The Mathematical Components project aims to demonstrate that formalised mathematical theories can, like modern software, be built out of components. To reach this objective, it proposes to develop a general platform for mathematical components, based on the Coq [29] SSReflect [45] extension, developed by Georges Gonthier to carry out the formalisation of the Four Colour Theorem [44].

We had the pleasure to work for six months in this team, contributing to the development of such modular components, meant to tackle in the long term the Feit-Thompson theorem [41]. The internship ended in march 2007 and the paper "A modular formalisation of finite group theory" [46] was published in August 2007.

The following section introduces the SSReflect extension, its main features and the base library. Section 2.2 details the design choices made to make the formalisation of finite groups run smoothly. Our main contributions are described in Section 2.2.3 and the following ones, where the encoding of finite and intensional sets is presented, together with all its implications in the definitions of tuples, function spaces and actions. Parts of the Section 2.1 and Section 2.2 are reworked extensions of [46]. In the last Section 2.3 some considerations on the approach, both methodological and technical, used in the Mathematical Components projects are made. Some links with the work the author made as a developer of the Matita interactive theorem prover are also analysed.

2.1 SSReflect

The SSReflect extension [45] offers a new syntax for the Coq proof shell and a bunch of libraries making use of *small scale reflection* in various respects.

Small scale reflection is so named because instead of using the computational aspect of the logic to run full-blown decision procedures or algebraic simplification algorithms ("big-scale" reflection, like the one use by the *ring* tactic [18]), it uses it to automate small menial operations, such as locally unfolding a function definition, often using directly the computational content of the objects under study and thereby avoiding the clumsy reification process. This technology was developed for the proof of the four colour theorem [44], where so many concepts were falling in this category.

To make this reflection work smoothly some tactics have been modified, for example making use of the refiner (type inference) subsystem of Coq, instead of the simpler type checker, to benefit from the canonical structures mechanism. As we will see in the following section, this allows some sort of modularity. Theorems can be proved on polymorphic data structures where the abstracted type is equipped with a decidable equality. Then concrete types can be linked with their equality decision function and its properties by the canonical structure mechanism, and the user is freed from the burden of providing additional informations when these lemmas are applied to a structure on the concrete type. Moreover, Coq tactics are not written to handle boolean propositions, thus a convenient way to switch between logical propositions and their boolean reflection was developed. In addition to that some tactics and best practices have been developed to make it easier to maintain proof scripts.

In this section, we describe the proof shell SSReflect provides, and comment the fundamental definitions present in the library and how modularity is carried out throughout the development.

2.1.1 The proof shell

The commands allowed in this new proof shell syntax aim at providing both more structure in the proof scripts and more control on the operations performed on terms. The main issue is to obtain proof scripts that are more robust to changes in the definitions: broken scripts are usually difficult to mend because failures occur too late after the place where incompatibilities should have been detected. Proof scripts written with the SSReflect extension have a very different flavour than the ones developed using standard Coq tactics. Many of Coq's primitive tactics, like **intro** or **inversion** are performing complex operations with a minimal input from the user. The SSReflect package promotes the use of a small, but compositional set of operations, with a sharp control on the occurrence at which they are performed.

Structuring scripts

Structuring a proof script helps for sure the reader to figure out the sketch of the informal proof described. Yet a robust structure is also useful at development time to detect as early as possible the changes implied in a replayed script by a previous modifications in the definitions involved. This may not be considered a frequent scenario, and in fact it is for small developments. But when the aim is a huge formalisation like the Feit-Thompson theorem, definitions have to be chosen with extreme care, and since there is no silver bullet, this can only happen by trial and error. Having robust scripts, easy to maintain, is the key point for experimenting approaches/encodings at a reasonable cost.

We can draw a parallel with software development, where the more a compiler is strict (for example using static typing) the more it is easy to change a data or function type. The compiler will spot all places where there is a misuse of such data or function.

The first structuring feature consists in *closing commands*. Detecting modifications needed in a script is eased if the proof of any subgoal should end with a tactic which fails if it does not solve it. The **done** and **by** tactics try to solve the current goal by trivial means (simplification, assumption already in the context,...) and fail if it does not succeed. They are both highlighted in bright red (using the Proof General[5] interface), with the purpose of better marking where subproofs end. The sequence: **by** t1 ;...; tn. is equivalent to t1 ;...; tn; **done**. but the former is to be preferred since prefixing the terminator better marks the ending line of the subproof. A best practice, used in a consistent way by the Mathematical Components team, is to indent subproofs longer than one line, ending them with the prefix **by** tactic.

The **done** tactic is implemented as an Ltac [36] tactic performing the following operations:

- Introducing Hypothesis
- Applying the **split** tactic (actually it is equivalent to constructor 1 in Coq)

- Tempting to solve the goal with the tactics: **trivial**, contradiction, discriminate, **assumption** and
- If the context contains a proposition P and its negation (actually P →False in Coq) conclude by inhabiting False using modus ponens.

Another way of highlighting the steps of a proof is to use forward/backward chaining commands. This affects the shape of the proof term constructed by abstracting some of its subterms. This declarative style is closer to the one of pen-and-paper proofs and can be freely mixed with the procedural one.

These commands differ from the available standard Coq tactics **cut** and **assert** by their internal implementation through abstraction rather than by a let-in binding: this approach seems to better interact with Coq's term comparison algorithms.

The possibility of mixing declarative and procedural proof styles has proven to be quite effective. In group theory many proofs begin defining a construction (like a group action or a particular set), but then the proof is performed reasoning in a backward fashion.

Bookkeeping

By bookkeeping commands, we mean operations which move things between the hypotheses and the goal, while changing their shape (by decomposition, generalisation, simplification).

A single command is used to **move** things between the context and the goal, complying with Coq standard intro-patterns. The latter allow to decompose by case analysis the objects that are introduced or generalised. The user can at the same time name the generated terms or clear useless hypotheses from the context.

For example, if the context contains a hypothesis H:∀ n, P n, the command **move**:(H 0) generalises (H 0) and the command **move**:(H 0) ⇒H0 puts a new hypothesis H0:P 0 in the context. If the goal is of the form A1 →A2 ∧A3 →A4, after the command:

```
move⇒ A1 [A2 A3] {H0}
```

the goal becomes A4, three hypotheses H1:A1, H2:A2 and H3:A3 are created. The hypothesis H0 is cleared from the context.

The switch //, that tries to solve goals with the **done** tactic, can be placed everywhere, even in between an intro pattern. Special names _ and → can be used to drop an hypothesis (introducing it without a name) or introduce an hypothesis, rewrite it in the goal and clear it.

It is again clear the purpose of providing a compact command to make scripts more robust. The commonly used **intros** tactic, introduces as many hypothesis as possible in the context, guessing names for them. The tactic works even if there are no hypothesis to introduce, avoiding an early detection of script breakage. Moreover the name guessed by the tactic depend over the type (sort) of the terms moved to the context, and the proof script that follows relays on these names. **move** insists in asking the user the names for the hypotheses, making it impossible that an H2 is later used instead of an H1 just because H1 changed its sort to Type and thus the name guessing algorithm named it X1 shifting H2 to H1.

Rewriting

Working with a decidable equality gives to equational reasoning a favoured status (see sections 2.1.2 and 2.1.3). The proposed extension of the rewriting command of standard Coq allows both better to control the occurrences to be rewritten, and to chain in a single command a list of rewrite steps. A rewrite step can either rewrite a rule, or fold/unfold a definition, or apply the standard **simpl** tactic (essentially $\beta\iota$ reduction), or try to solve the current goal by trivial means. As soon as it makes some sense, each rewrite step can be given an orientation flag, and/or a list of occurrences, and/or a pattern, and/or a multiplier.

```
rewrite !lem1 {3}[x ∗ _]lem2 2!lem3 /= −?lem4 /mydef //.
```

- rewrites lem1 in the current goal as many times as possible (but at least one);
- then, rewrites lem2 at the third occurrence fitting the pattern [x ∗ _] in all the subgoals generated by the previous rewriting;

- then, rewrites lem3 exactly 2 times in all the subgoals generated by the previous rewriting;
- then, simplifies in all the subgoals generated by the previous rewriting;
- then, rewrites lem4, right to left, as many times as possible in all the subgoals generated by the previous rewriting (even zero times is accepted);
- then, unfolds, if possible, the definition mydef in all the subgoals generated by the previous rewriting;
- then closes the subgoals generated by the previous rewriting that can be trivially solved.

When most of the lemmas are stating equalities, the possibility to chain rewriting is extremely handy. All operations that are usually performed between single rewriting step have a compact counterpart in SSReflect: the possibility to specify an occurrence resembles the standard Coq pattern tactic invocation; while /name and /= perform unfolding and simplification. The switch // to close trivial subgoals and ? to perform optional rewriting is essential when a guarded equality is used, and the premise obligation can be quickly discharged (by means of another rewriting for example) without breaking the rewrite chain. Having symbolic names for trivial operations like simplification, partially hides them, giving more importance to the names of the lemmas involved in the proof.

Case analysis on dependent types

Even if dependent types are not heavily used in SSReflect, there is a special switch to handle case analysis over dependent types. The classical example is equality

```
Inductive eq (A:Type) (x:A) : A →Prop := refl_equal : eq A x x
```

with the associated elimination principle

```
eq_ind : ∀A x (P : A →Prop) (p1 : P x) y (e : x = y) (P y)
```

When we do case analysis over a proof of an equality we have to help Coq in inferring how to type the branches of the pattern matching. As an example we take a lemma needed for proving the irrelevance of proofs of an equality between types over which equality is decidable [53]

```
Lemma stepK : ∀(d : eqType) (x : d) E,
  refl_equal  x = eq_ind x (fun y  ⇒y = x) E x E.
Proof.
move⇒ d x E.
case: {2 3 4 5 7 8}x / E.
 reflexivity .
Qed.
```

The tactic **case** produces a pattern matching over E. In CIC, inferring a common type for all branches is in general undecidable. As a workaround, every pattern matching construct over an inductive type T is equipped with a typing function f. If T is an inductive type with some parameters params and has n constructors K_i each of them taking args_i parameters, the typing function f is expected to return the type of the i-th branch if applied as follows: f params ($\mathrm{K}_i\mathrm{args}_i$). In this case the pattern match construct has to be equipped with the following function

```
f := fun (r : d) (e : x = r) ⇒refl_equal r = eq_ind x (fun y ⇒y = r) e r e
```

since eq has one parameter (abstracted with name r here). Unfolding the notation = and making all types explicit we obtain

```
f := fun (r : d) (e : eq d x r) ⇒
 @eq (@eq d r r) (@refl_equal d r) (@eq_ind d x (fun y : d ⇒eq d y r) e r e)
```

The list of occurrences specified to the **case** tactic is the list of occurrences of x in the goal that have been bound to r in f.

The type of E, eq d x x, can be inhabited by just one term, refl_equal d x that fixes the parameter of eq to x. f is thus applied to x and refl_equal d x, obtaining the following type

```
@eq (@eq d x x) (@refl_equal d x)
    (@eq_ind d x (fun y : d ⇒eq d y x) (@refl_equal d x) x
    (@refl_equal d x))
```

That simplifies to refl_equal x = refl_equal x and thus can be inhabited using the reflexivity tactic.

Searching the library

Standard Coq provides some functionalities to search the library of already proved theorems. SearchPattern [13] is easy to use, but sometimes not flexible enough. The integration with Whelp (a search engine briefly described in Section 4.3.1) was not complete at the time of the internship (the author helped the Coq team to set up a Whelp server internal to INRIA). Moreover the SSReflect library was not yet released to the public and thus not indexed by the Whelp server in Bologna. SSReflect implements a **Search** command that resembles the **match** query of whelp: it is possible to specify a list of constants that may appear in the conclusion or hypothesis of the searched lemma, and also fix an head constant for the conclusion.

For example, to look for all equalities that use both the orbit and the stabiliser constant in their statements the following command could be used:

```
Search eq [orbit stabilizer ].
```

No feature of SSReflect has been used more frequently by the author of this manuscript after it was implemented (that unluckily happened after few months of the internship where passed).

2.1.2 Small scale reflection

The Coq system is based on an intuitionistic type theory, the calculus of inductive constructions [93, 73, 30]. Yet when the theory to be formalised inside the system is classical, developing proofs may not go as smooth as expected. The SSReflect extension proposes a workaround to recover the ease of classical reasoning in Coq when it is allowed by the theory.

In the Coq system, logical propositions are represented by types having the sort Prop. Of course, this sort does not enjoy the excluded middle principle : the proposition $\forall P : Prop, P \vee \neg P$ is not provable. On the other hand, bool is an inductive datatype with two constructors true and false, and the pattern matching on its constructors allows classical reasoning on predicates defined as boolean functions. The boolean datatype can be injected into this Prop sort thanks to the following function:

```
Coercion is_true := fun b : bool →b = true.
```

The coercion mechanism of Coq realises this injection inside the system. It automatically inserts applications of the is_true function when a boolean needs to be changed into a proposition for a term to be well typed. By default, these applications are hidden to the user at display time for sake of clarity.

Another key issue about boolean predicates is their computational behaviour: two such predicates are logically equivalent if and only if their values are equal. Hence *rewriting* becomes a way to handle equivalent statements. Anyway the Prop level is still needed when one needs to perform primitive tactics on the boolean statements: case analysis, generalisation, ...

In practice, going back and forth between booleans and their coerced version needs to be easy and convenient. The relation between a boolean predicate and a proposition is realized by the reflect inductive predicate: (reflect P b) means that (is_true b) and P are logically equivalent.

```
Inductive reflect (P : Prop) : bool →Type :=
  | Reflect_true : P →reflect P true
  | Reflect_false : ¬P →reflect P false .
```

For instance we can prove the following lemma:

```
Lemma andP : ∀b1 b2, reflect (b1 ∧b2) (b1 && b2).
```

where && stands for the boolean conjunction, and ∧ for the logical one. Two coercions have been inserted to make b1 ∧ b2 a well-typed term. Such a lemma, bridging the gap between boolean and logical definitions is called a *view* lemma. By convention the name of these view lemmas are postfixed with a capital P.

If we are to prove a goal of the form (x == y) && (z == t) →G, where == stands for the computational equality between booleans, the tactic **move**/andP; **case** performs the conversion and the destruction, changing the goal into x == y ∧ z == t →G. The use of an intro-pattern allows to introduce and name the hypothesis created **move**/andP ⇒[Eq1 Eq2].

Views can be combined with most tactics, for example the previous operation can also be performed with **case**/andP ⇒Eq1 Eq2.

Suppose now that we have to prove the goal x == y && z == t. In order to split this goal into two subgoals, we use a combination of two tactics: **apply**/andP; **split**. The first tactic reflects the boolean conjunction into a logical one, the second tactic can then perform the splitting. In fact we seldom use the /andP view in this way, since directly *rewriting* boolean hypotheses (like (x == y) = true) in such a goal will ultimately simplify it to z == t.

Another possible usage of views is to prove equalities by means of a double implication. Assuming two unary boolean predicates p1 and p2, one could attack a goal like ∀ x, p1 x = p2 x with

```
apply/view_p1/view_p2.
```

obtaining two goals: P1 x →P2 x and P2 x →P1 x where:

```
Lemma view_p1 : ∀ x, reflect (P1 x) (p1 x)
Lemma view_p2 : ∀ x, reflect (P2 x) (p2 x)
```

The /_/_ syntax is convenient way of composing together the following lemmas:

```
Lemma introTF: ∀ (P2 : Prop) (c b : bool), reflect P2 c →
  (if b then P2 else ¬P2) →b = c
Lemma equivPif: ∀ (P1 P2 : Prop) (b : bool), reflect P1 b →
  (P2 →P1) →(P1 →P2) →if b then P2 else ¬P2
```

Views also provides a convenient way to swap between several (logical) characterisations of the same (computational) definition. One can prove a view lemma per interpretation of the definition. Each different view applying to a same boolean term gives indeed one of its logical interpretations.

A trivial example of this multiple interpretations possibility is given by the view lemmas associated with n-ary boolean connectives. Fore instance, if andb3 b1 b2 b3 is defined as andb b1 (andb b2 b3), and and3 as its logical counterpart, then we can prove the view lemma:

```
Lemma and3P :
  ∀b1 b2 b3, reflect (and3 b1 b2 b3) (and3b b1 b2 b3)
```

This lemma allows to decompose a triple boolean conjunction in a single operation : a goal of the form (and3b b1 b2 b3 →G) is transformed into (b1 →b2 →b3 →G) by applying the **case**/and3P tactic. But it is also possible to use the **case**/andP tactic in this case, and this will lead to the goal (b1 →b2 && b3 →G).

2.1.3 Libraries for decidable and finite types

SSReflect provides as a standard library a huge set of lemmas underlying the formal proof of the Four Colour theorem. These ones build a hierarchy of structures to handle conveniently the types equipped with a decidable equality, and a substantial toolbox to work with finite sets.

The minimal requirements for the type of object to fit this framework is to be equipped with a decidable relation which is Leibniz equality compliant. Such an object is called an eqType structure.

```
Structure eqType : Type := EqType {
  sort :> Type;
  eq : sort →sort →bool;
  eqP : ∀x y, reflect (x = y)(eq x y)
}.
```

The :> symbol declares sort as a coercion from an eqType to its carrier type.

In the type theory of Coq, the only relation that can be handled by rewriting is the primitive (Leibniz) equality. When an other equivalence relation is the intended notion of equality on a given type, the user usually needs to use the setoid [10, 25] workaround. But setoid rewriting does not have the full power of primitive rewriting. An eqType structure not only assumes the existence of a decidable equality: the eqP requirement injects this equality into the Leibniz one, and makes it a *rewritable* relation.

Finite sets can reasonably be formalised on top of eqType structures, since they are equipped with a natural decidable equality which will satisfy eqP, for instance as soon as the set is described by a non parametric inductive type. An elementary example of such a structure can be built with the type of Peano natural numbers. We will even declare this structure, called nat_eqType as a **Canonical Structure**. This feature of standard Coq allows to solve equations involving implicit arguments. Namely, if the type inference algorithm needs to infer an eqType structure on the type nat, it will choose as a default choice the nat_eqType type. Another such example is given by the bool_eqType structure. By enlarging the set of implicit arguments Coq can infer, canonical structures ease a lot the handling of the hierarchy of structures.

An eqType structure enjoys proof-irrelevance for the equality proofs of its elements: every such equality proof is convertible to a reflected boolean test.

```
Lemma eq_irrelevance :
  ∀(d : eqType) (x y : d) (E E' : x = y), E = E'.
```

and in particular for the bool_eqType structure. Note that here bool is lifted to bool_eqType by the canonical structure mechanism of Coq.

```
Lemma bool_irrelevance :
  ∀(x y : bool) (E E' : x = y), E = E'.
```

An eqType structure should not be understood as a set itself. It merely gives a domain, in which some sets involved in the formalisation will take their elements.

Extensional sets are represented by their characteristic function and defining set operations is done by providing the corresponding boolean functions.

```
Definition set (d : eqType) := d →bool.
```

The next step is to build lists, whose elements belong to a certain eqType structure. The terms list and sequence will be used with the same meaning in the following. The decidability of the comparison between elements is central to the programming of most basic operations on lists like membership and indexing. A declared coercion between the type of lists seq, and the class of functions identifies a sequence to the (finite) set of its elements.

```
Definition setU1 (d : eqType) (x : d) (a : set) : set d := fun y ⇒(x == y) || a y.
Fixpoint mem (d : eqType) (s : seq d) : set d :=
  if s is Adds x s' then setU1 x (mem s') else set0.
Coercion mem : seq ↦set.
```

With setU1 x s we build the set corresponding to the union of the set s and the singleton set containing only x. Lists are the cornerstone of the definition of finite sets. A finType structure can be seen as a list of non redundant elements of a certain eqType structure.

```
Structure finType : Type := FinType {
  sort :> eqType;
  enum : seq sort;
  enumP : ∀x, count (set1 x) enum = 1
}.
```

The utility function count returns the number of elements in a sequence that make a boolean predicate evaluate to true. set1 is a synonym for the computational equality of an eqType: here it could be unfolded (making notation explicit) to: eq sort x. A bunch of cardinality lemmas is proved in this library, for example concerning the combination with set operations like union and intersection:

```
Lemma cardUI : ∀A B,
  card (A ∪ B) + card (A ∩ B) = card A + card B.
```

Since sets are characteristic function, the image of a set by an application is simply the composition of the function and the set. If f is an application between two finType structures, then:

```
Definition injective : Prop := ∀x y : B, f x = f y →x = y.
Lemma card_image : ∀(d d' : finType) (f : d →d') (A : set d),
  injective f →card (image f A) = card A.
```

We are not going to detail the implementation of card and image, but they are clearly implementable with recursive functions since their domains are (subset of) a finite domain, actually a finType, that embeds the canonical enumeration. Moreover elements in a finType are also elements of an eqType, thus the mem coercion can adjust the type of a sequence (namely the result of image) to a **set**.

As one may expect, lists whose elements lie in an eqType can be equipped with a set of lemmas much richer that the standard Coq polymorphic lists. The SSReflect library has around 300 lemmas concerning sequences, while standard Coq has less that an half.

These lemmas are abstracted over a generic eqType, thus they can be used on any type for which a boolean comparison function can be proved Leibniz compatible. Moreover the canonical structure mechanism of Coq, allows the user to apply lemmas to sequences of the base (non eqType) type (like bool or nat and let the system infer the canonical, Leibniz compatible, comparison function needed by the lemma. The same facility applies to finType.

2.2 Finite group theory

The author contributed to the formalisation described in this section during his internship in the Mathematical Components team at the INRIA-Microsoft Research Lab in Orsay, Paris. The work has been published in the paper "A modular formalisation of finite group theory" [46].

2.2.1 Groups domains and quotients

Just like eqType structures were introduced before defining sets, we introduce a notion of (finite) *group domain* which is distinct from the one of groups. It is modelled by a finGroupType record structure, packaging a carrier, a composition law and an inverse function, a unit element and the usual properties of these operations. Its first field is declared as a coercion to the carrier of the group domain, that itself can be coerced to a Type, using the sort coercions of the finType and eqType structures.

```
Structure finGroupType : Type := FinGroupType {
  element :> finType;
  unit : element;
  inv : element →element;
  mul : element →element →element;
  unitP : ∀x, mul unit x = x;
  invP : ∀x, mul (inv x) x = unit;
  mulP : ∀x1 x2 x3, mul x1 (mul x2 x3) = mul (mul x1 x2) x3
}.
```

In the group library, a first category of lemmas is formed by the identities being valid on the whole group domain. For example:

```
Lemma invg_mul : ∀x1 x2 : elt, (x2 ∗ x1) ⁻¹ = x1 ⁻¹ ∗ x2 ⁻¹ .
```

One can also already define operations on arbitrary subsets of a group domain. If A is such a subset, we can define for instance:

```
Definition conjg (g : finGroupType) (x y : g) := x ⁻¹ ∗ y ∗ x
Definition rcoset A x := {y, y ∗ x ⁻¹ ∈A}.
Definition sconjg A x := {y, y ^ x ⁻¹ ∈A}. (∗ denoted A:^x ∗)
Definition normaliser A := {x, (A :^ x) ⊂ A}.
```

The notation {x,P} can be seen as (fun x ⇒P) for the moment, but when we will switch to intensionally represented sets, it will slightly change.

A second category of lemmas take inventory of the properties of these operations, requiring only group domain *subsets*.

These definitions are as often as possible defined as boolean predicates, combining the ones that are already available in the libraries. This leads to definitions that may look less intuitive at first sight. The set of point-wise products of two subsets of a group domain is for instance defined as:

```
Definition smulg A B := {xy, ~disjoint {y, rcoset A y xy} B}.
```

A *view* lemma gives the natural characterisation of this object, where A :∗: B stands for (smulg A B) :

```
Lemma smulgP : ∀A B z,
  reflect (∃x y, x ∈ A ∧y ∈ B ∧z = x ∗ y) (z ∈ A :∗: B).
```

Finally, a *group* is defined as a boolean predicate, satisfied by subsets of a given group domain that contain the unit and are stable under product.

```
Definition group_set A := 1 ∈A && (A :∗: A) ⊂ A.
```

It is very convenient to allow the possibility of attaching in a canonical way the proof that a set has a group structure. Hence groups will themselves be declared as structures:

```
Structure group(elt : finGroupType) : Type := Group {
  set_of_group :> set elt ;
  set_of_groupP : group_set set_of_group
}.
```

We will declare a canonical structure of groups for objects like the normaliser of a group or the kernel of a morphism so that they can be displayed as their set carrier but benefit from an automatically inferred proof of a group structure when needed. An example showing this technique is provided at the end of the section.

Two groups will share the same group domain, if they share the type of their elements, the operations and unit. A new finGroupType construction is needed if and only if one (which in general means all) of these ingredients vary.

The boolean definition of groups enjoys the proof-irrelevance of such predicates. Proving that two groups are *equal* hence boils down to proving the equality of their carriers as sets in the same finType:

```
Lemma inj_set_of_group :
  ∀elt: finGroupType, injective (set_of_group elt ).
```

Performing case analysis over the two groups to be introduced, we obtain the following goal:

```
elt : finGroupType
s1, s2 : set elt
s1G : group_set (elt:=elt) s1
s2G : group_set (elt:=elt) s2
E : s1 = s2
=====================================
Group (elt:=elt) (set_of_group:=s1) s1G = Group (elt:=elt) (set_of_group:=s2) s2G
```

After the rewriting of the E hypothesis in s1G and in the current goal, we need to prove that :

```
Group (elt:=elt) (set_of_group:=s2) s1G = Group (elt:=elt) (set_of_group:=s2) s2G
```

under the assumption that the terms s1G and s2G are proofs of the same boolean predicate (group_set (elt:=elt) s2). We can then conclude by boolean proof-irrelevance. The script realising this proof is the following:

```
case⇒ s1 s1G; case⇒s2 s2G /= E; rewrite E in s1G ⊢ ∗.
by rewrite (bool_irrelevance s1P s2P).
```

This proof irrelevance combined with the uniformity of group domains results in a convenient framework to deal with different characterisations of a same mathematical object. The reader may object that in an intensional type theory like CIC, equations like E are in general not provable, since the sides of the equality are functions and the extensionality rule for functions is not provable in CIC. In sections 2.2.2 and 2.2.3 an evidence of this problem and the adopted solution are discussed.

Thanks to the above lemma and to the fact that they are sets on the same finType domain, proving that the kernel of a morphism is a trivial *group* consists in proving that the kernel set is equal to the singleton of the unit.

Now come the last kind of lemmas in the library, which state group properties. For example, if H is a group, then:

```
Lemma groupMr : ∀x y, x ∈H →(y ∗ x) ∈H = y ∈H.
```

In the above statement, the equality stands for Coq standard equality between boolean values, since membership of H is a boolean predicate.

The canonical structure mechanism came again handy, since we declared groups as canonical. For example the normaliser of a subset of a group can be proven to be a group.

```
Variable elt : finGroupType
Theorem group_set_normaliser : ∀A : set elt, group_set (normaliser A).
Canonical Structure group_normaliser := Group group_set_normaliser.
```

The lemma groupMr can be rewritten in the following statement.

```
elt : finGroupType
A : set elt
Hx : x ∈ normaliser A
Hy : y ∈ normaliser A
=====================================
(x ∗ y) ∈ normaliser A
```

The group_set_normaliser lemma will be inferred by Coq, changing the goal to

```
x ∗ y ∈ Group (normaliser A) (group_set_normaliser A)
```

2.2.2 Quotients and the need for intentional sets

Let H and K be two groups in the same group domain. The group H is *normal* in K, denoted $H \lhd K$ if all elements of K commute with all elements in H. More formally:

$$H \lhd K \overset{\text{def}}{=} \forall h \in H, k \in K, k^{-1} * h * k \in H$$

Given the definition of normaliser of the previous section:

```
Definition normal H K := K ⊂ (normaliser H).
```

Given a group H, in a group domain elt, all the well-formed quotients of the form K / H will share the same group law, and unit. Such a quotient is well defined as soon as $H \lhd K$, and the group operation for the quotient is the multiplication of H-cosets. The largest quotient on can build from a given group H is $N(H)/H$, where $N(H)$ is the normaliser of H.

In particular, left and right H-cosets are identical for all the elements of K. Hence they will be called *cosets*. The set of these H-cosets of element in K is denoted K/H, and one can define a product law on it which equips the set with a group structure.

This construction requires anyway to build a new group domain structure since the quotient group has a new domain and group operation. All the quotients of the form $\cdot/H$ will share the same carrier, the H-cosets, and the same operations and unit. Since the largest possible quotient is $N(H)/H$, all the other well-defined quotients are subsets of this one.

If H is a group in the group domain elt, let Hquo be the finGroupType giving the group domain of the quotient by H. It is a new type, depending on elt and H.

Once again, we carefully stick to first order predicates to take as much benefit as possible from the canonical structure mechanism. If necessary, side conditions are embedded inside definitions by the mean of boolean tests. Like this, we avoid having to add pre-conditions in the properties of these predicates to insure well-formedness. The definition of cosets makes no restriction on its arguments:

```
Definition coset (A : set elt) (x : elt) := if (x ∈ (normaliser A)) then A :* x else A
```

The set of cosets of an arbitrary set A is the image of the whole group domain by the coset operation. Here we define the associated sigma type:

```
Definition cosets (A : set elt):= image (coset A) elt.
Definition cosetType (A : set elt):= eq_sig (cosets A).
```

where eq_sig is the type constructor for the sigma type associated to a **set**. This cosetType type can be equipped with canonical structures of eqType and finType and elements of this type are intentional sets.

The quotient of two groups of the same group domain can *always* be defined:

```
Definition A/B := image (coset_of B) A.
```

where coset_of : elt →(cosetType A) injects the value of (coset A x) in (cosetType A). Thanks to the internal boolean test in coset, A/B defines in fact $[A \cap N(B)]/B$.

When H is equipped with a group structure, we define group operations on (cosetType H) thanks to the following properties:

```
Lemma cosets_unit : H ∈(cosets H).
Lemma cosets_mul : ∀Hx Hy : cosetType H, (Hx :*: Hy) ∈(cosets H).
Lemma cosets_inv : ∀Hx : cosetType H, (Hx :^-1) ∈(cosets H).
```

where A $:^{-1}$ denotes the image of a set A by the inverse operation. Group properties are provable for these operations: we can define a canonical structure of group domain on cosetType, depending on an arbitrary group object. Canonical structures of *groups*, in this group domain, are defined for every quotient of two group structures. A key point in the readability of statements involving quotients is that the ./. notation is usable because it refers to a definition independent of proofs; the type inference mechanism will automatically find an associated group structure for this set when it exists.

Defining quotients has also been a place where we had to rework our formalisation substantially using intensional sets instead of sets defined by their characteristic function. In the library of finite group quotients, there are two kinds of general results. The first one states *equalities* between quotients, like the theorems about the kernel of quotient morphism. The second, often heavily relying on properties of the first kind, builds isomorphisms between different groups, i.e. groups having distinct carriers (and hence operations). For example, this is the case for the so-called three fundamental isomorphism theorems. The initial version of the quotients was using sets defined by their characteristic function. Having sets for which function extensionality does not hold had forced us to use setoid. For theorems with types depending on setoid arguments, especially the ones stating equalities, we had to add one extensional equality condition per occurrence of such a dependant type in the statement of the theorem in order to make these theorems usable. The situation was even worse since, in order to apply one of these theorems. The user had to provide specific lemmas, proved before-hand, for each equality proof. This was clearly unacceptable if quotients were to be used in further formalisations.

2.2.3 Function graphs and intensional sets

As mentioned in the previous section, defining set as CIC functions is handy but makes it almost impossible to prove that two sets are equal. While it is still possible to prove that they have the same elements, the extensionality rule for functions is not provable in CIC, and thus it is not possible to rewrite sets with the primitive equality. The setoids machinery ameliorates this problem, but still some extra conditions have to be proved, making statements bigger and harder to apply.

Although combining CIC with the extensionality axiom for function results in a consistent system, this time axioms can be avoided at all. Exploiting the fact that the domains which we are interested in are usually finite, functions can be represented with their graph. The following type declaration packs together a sequence of elements of type d2, the codomain of the function, and a proof that the length of the sequence is equal to the cardinality of the domain d1.

```
Variables (d1 : finType) (d2 : eqType).
CoInductive fgraphType : Type :=
  Fgraph (val : seq d2):(size val) = (card d1) →fgraphType.
```

The use of a coinductive type instead of a common record, actually coded as single constructor inductive type, inhibits the, although automatic, generation of

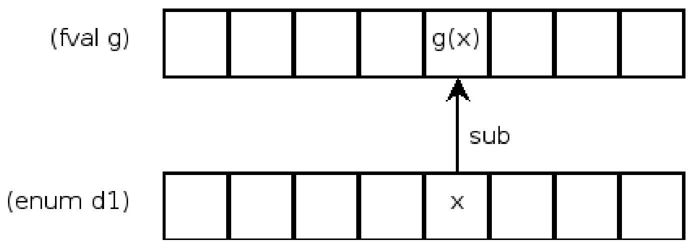


Figure 2.1: Using a function graph as a function

the elimination principle for fgraphType. This is done on purpose: since there is no recursion in this data type we don't need a recursor, we only need pattern match to extract the sequence or the proof if needed.

With this representation of functions, and the coercion fun_of_fgraph we obtain functions whose extensional equality implies the intentional one (lemma fgraphP), allowing to rewrite them. The coercion fun_of_fgraph (see Figure 2.1) takes a graph g and an element x of the domain, finds the position at which x occurs in the canonical enumeration of the domain and returns the element of g occupying that position (notice that the graph and the domain canonical enumeration have the same length).

```
Definition fun_of_fgraph (g : fgraphType d1 d2) :=
  fun x ⇒sub (fgraph_default x g) (fval g) (index x (enum d1)).
Coercion fun_of_fgraph : fgraphType ↦Funclass.
Lemma fgraphP: ∀(f g : fgraphType d1 d2), ∀x, f x = g x ↔f = g.
```

Here fval if the first projection of fgraphType, while index x (enum d1) returns the position of x in the canonical enumeration of the finite type d1. The sub operator selects the i-th element of the sequence s if i doesn't exceeds s length. The technical lemma fgraph_default exhibits a default element in d2, given an element x of type d1 and a function graph g (whose length cannot be zero since x proves that d1 is not empty).

```
Lemma fgraph_default : ∀d1 d2, d1 →fgraphType d1 d2 →d2.
```

To create a function graph given a regular function, we can use some standard tools already available in the SSReflect library, namely maps and size_maps. The former maps a function over a sequence, while the latter is a lemma, stating that maps does not alter the length of the mapped sequence. The fgraph_of_fun operator composes maps and size_maps allowing to obtain an intentional function writing fgraph_of_fun (fun x ⇒...).

```
Definition fgraph_of_fun (f : d1 →d2) := Fgraph (size_maps f (enum d1)).
```

We can then form the eqType and finType of fgraphType. The former needs the following lemma that states that comparing function graphs boils down to comparing the two sequences.

```
Variables (d1 :finType) (d2 : eqType).
Lemma fgraph_eqP : reflect_eq (fun f1 f2 ⇒fval f1 == fval f2).
Proof.
move⇒ [s1 Hs1] [s2 Hs2]; apply: (iffP eqP) ⇒[/= Hs1s2|→ //].
by rewrite Hs1s2 in Hs1 ⊢ ∗; rewrite /= (nat_irrelevance Hs1 Hs2).
Qed.
```

The first **move** command introduces the two function graphs we are comparing, breaking them in their sequences s1 and s2 and the proofs that these sequences have the right length. After the application of (iffP eqP) the following coimplication remains to be proven:

```
fval (Fgraph s1 Hs1) = fval (Fgraph s2 Hs2) ↔Fgraph s1 Hs1 = Fgraph s2 Hs2
```

Proving the right-to-left part is trivial and the obligation is closed in the first line. To prove the remaining goal, we again benefit of the proof irrelevance on equality proofs over a decidable type like nat.

This lemma allows to declare the eqType of function graphs, that is declared to be a canonical structure so that the system can infer fgraph_eqP when an fgraph is used as an eqType.

```
Canonical Structure fgraph_eqType := EqType fgraph_eqP.
```

To define the finType of function graphs, we have to exhibit an enumeration of all function graphs given the domain and codomain (when they are both finite). The recursive functions to generate the enumeration are written on simple sequences and lifted to function graphs later:

```
Variable d : finType.
Let seq2 x := seq (seq_eqType x).
Let multes (e : d) (s : seq2 d) := maps (Adds e) s.
Let multss (s : seq d) (s1 : seq2 d) := foldr (fun x acc ⇒(multes x s1) @ acc) [] s.
Definition mkpermr length := iter length (multss (enum d)) [[]].
```

Here Adds is the curryfied constructor of sequences (usually Cons); we used [] and @ as a notation for the empty sequence and concatenation. The functions iter and foldr are pretty standard.

In the following, d1 and d2 are variables of type finType and fgraphType has to be considered instantiated on d1 as source and d2 as target.

To lift the enumeration of sequences of size (card d1) on a finite alphabet d2, we have to lift it to an enumeration of function graphs.

```
Definition infgraph (s : seq d2) : option fgraphType :=
 if size s =P (card d1) is Reflect_true Hs
 then Some (Fgraph Hs) else None.
```

The notation (**if** c **is** K **then** a **else** b) is interpreted as a pattern match in which the second branch matches _ (i.e. nothing is bound in that branch). The notation =P hides the application of the eqP field of an eqType. In this case size returns an inhabitant of nat, that is lifted to nat_eqType by the canonical structures mechanism, thus matched value has type

```
size s =P (card d1) : reflect (size s = card d1) (eq nat_eqType (size s) (card d1))
```

The function infgraph injects regular sequences into function graphs if they have the right size. The infgraph_spec will be used to specify the infgraph function, that will be mapped on the result of mkpermr discharging empty (None) elements.

```
CoInductive infgraph_spec (s : seq d2) : option fgraphType →Type :=
 | Some_tup u : size s = (card d1) →fval u = s →infgraph_spec s (Some u)
 | None_tup: ¬(size s == (card d1)) →infgraph_spec s None.
Lemma infgraphP : ∀s, infgraph_spec s (infgraph s).
```

The function infgraphseq filters out empty elements and is the real injection from the enumeration of sequences to the enumeration of function graphs. To obtain the finType of this enumeration the finfgraph_enumP lemma has to be proved.

```
Definition infgraphseq : seq2 d2 →seq fgraph_eqType :=
 foldr (fun (s:seq d2) ts ⇒if infgraph s is Some u then u :: ts else ts) [].
Definition finfgraph_enum := infgraphseq (mkpermr d2 (card d1)).
Lemma finfgraph_enumP : ∀u, count (set1 u) finfgraph_enum = 1.
Canonical Structure fgraph_finType := FinType finfgraph_enumP.
```

If d2 is instantiated with bool_eqType, the fgraph_finType presented above, can be used to represent finite sets as their characteristic (and intentionally represented) function. These objects, called setType can be rewritten using the standard equality elimination principle.

```
CoInductive setType : Type :=
  Sett : fgraphType d bool_finType →setType.
Definition sval (s : setType) := match s with Sett g ⇒g end.
Lemma isetP: ∀(a b : setType), ∀x, a x = b x ↔a = b.
Lemma s2f : ∀f x, iset_of_fun f x = f x.
Canonical Structure set_eqType := EqType (can_eq can_sval).
Canonical Structure set_finType := FinType (can_uniq can_sval).
```

The lemma isetP is provable and shows that we reached our objective of having sets that validate the extensionality law for functions. Thus sets can be proved equal comparing their elements but still be rewritable with the standard Leibniz equality. The need for a type tag Sett and the lemmas can_eq and can_uniq will be discussed in details in Section 2.3.1.

The lemma s2f is extensively used in the whole Mathematical Components files to switch from the intensional to extensional representation of sets. The type of the lemma hides the fun_of_iset coercion, analog to fun_of_fgraph. For an usage example look at the following example stating that the singleton intensional set on x contains x:

```
Lemma iset11 : ∀ G:finType, ∀ x:G, {: x} x.
Proof. move ⇒ x; rewrite s2f; exact: eq_refl. Qed.
```

The notation {: _} is defined as follows (precedences and associativity are omitted):

```
Notation "{ x , P }" := (iset_of_fun (fun x ⇒P)).
Definition iset1 x : setType G := {y, x==y}.
Notation "'{:' x }" := (iset1 x).
```

The statement is thus understood by Coq as (displaying coercions):

```
Lemma iset11 : ∀ G:finType, ∀ x:G,
  is_true ( fun_of_iset ( iset_of_fun (fun x ⇒x == y)) x).
```

Rewriting with fun_of_iset we obtain is_true ((fun x ⇒x == y) x) and we can thus let the system β-reduce the statement and close the obligation using the lemma eq_refl stating that the computational equality over an eqType rewrites to true if it is comparing the same variable.

The "exact:" tactic is slightly more powerful then the standard exact tactic: it refines the given term and calls **done** to close any goals left open.

2.2.4 Function spaces

The function graph construction does not lead only to the definition of finite sets that play nicely with Leibniz equality, but is flexible enough to encode many other mathematical objects.

Here we present the encoding of function spaces. The set of functions from a finite domain d1 to a finite codomain d2 is expressed with the binary boolean function a : d1 →d2 →bool. This is general enough to describe the space of total function to a subset of the codomain and the space of partial function (from a subset of the domain). Partial function spaces are identified with an element of the codomain they must map to when the input is outside the domain.

The function (actually a relation) a is not represented as a function graph, but as a CIC function since there is no need to use rewriting on such objects here. Anyway, what follows could be defined using function graphs with minor modifications.

```
Variables d1 d2 : finType.
Variable a : d1 →set d2.
Fixpoint prod_seq (s1:seq d1) {struct s1} : set (seq d2) :=
 match s1, s2 with
 | Adds x1 s1', Adds x2 s2' ⇒a x1 x2 && prod_seq s1' s2'
 | _, _ ⇒true
end.
Definition fgraph_prod (u : fgraph_eqType d1 d2) : bool := prod_seq (enum d1) (fval u).
```

The boolean predicate fgraph_prod is specified by the following proposition, stating that the functions inside the function spaces are subset (seen as relations) of a.

```
Lemma fgraph_prodP: ∀u : fgraphType d1 d2, reflect (∀x, a x (u x)) (fgraph_prod u).
```

Some result about the cardinality of function spaces have been proved. All of them are particular case of the following lemma:

```
Lemma card_fgraph_prod:
 card fgraph_prod = foldr (fun i m ⇒card (a i) ∗ m) 1 (enum d1).
```

Total function spaces are defined instantiating a with a relation that covers the whole domain. With a2 we identify a subset of the codomain and with (fun _ x ⇒a2 x) the relation for fgraph_prod.

```
Variable a2 : set d2.
Definition tfunspace : set (fgraph_eqType d1 d2) := (fgraph_prod (fun _ ⇒a2 )).
```

```
Lemma tfunspaceP : ∀u : fgraphType d1 d2, reflect (∀x : d1, a2 (u x)) (tfunspace u).
Lemma card_tfunspace : card tfunspace = (card a2) ^ (card d1).
```

Partial function spaces are built with a restriction of the domain a1 and a restriction of the codomain a2. With a2' we identify the singleton set containing only y0 to which function should map elements outside a1.

```
Variable y0 : d2.
Variable a1 : d1 →bool.
Variable a2 : d2 →bool.
Let a2' := {: y0}.
Definition pfunspace :=
 @fgraph_prod d1 _ (fun i ⇒if a1 i then a2 else a2').
Lemma card_pfunspace: card pfunspace = (card a2) ^ (card a1).
```

To characterise the boolean predicate pfunspace we define the support boolean predicate that checks if an element in the domain is mapped to y0. The predicate sub_set can be unfolded to ∀s1 s2 x. s1 x = true →s2 x = true.

```
Definition support g (x : d1) : bool := g x != y0.
Lemma pfunspaceP : ∀g : fgraphType d1 d2,
  reflect (sub_set (support g) a1 ∧ sub_set (image g a1) a2) (pfunspace g).
```

Function spaces will be used, although not heavily, in the proof of the Cauchy theorem in Section 2.2.7.

2.2.5 Tuples and rotations

Homogeneous tuples are a common mathematical object and, for example, will be used in the proof of the Cauchy theorem presented in Section 2.2.7. An n-tuple could be encoded directly as sequence of length n, but since we already have the same machinery for function graphs we could try to reuse it. A tuple of elements living in a type T can be seen as a function from a finite set of cardinality n to T.

The standard SSReflect library already provides the finite type of an initial segment of the natural numbers. The utility function (iota n m) generates the sequence of natural numbers starting from n of length m. (subfilter P s), where P is a boolean predicate, filters a sequence using P also injecting elements in the Σ-type Σx: nat. P x = true.

```
Variable n : nat.
Definition ord_enum := subfilter (fun m ⇒m < n) (iota 0 n).
Lemma ord_enumP : ∀u, count (set1 u) ord_enum = 1.
```

To complete the encoding we need an easy way of generating an element in the domain of ordinals make_ord and declare the type of n-tuples of elements in d with fgraphType (ordinal n) d.

```
Definition make_ord : ∀m, m < n →ordinal :=
 fun m H ⇒EqSig H.
Definition ordinal := FinType ord_enumP.
Variables (d : finType) (n : nat).
Definition tupleType := fgraphType (ordinal n) d.
```

Retrieving the n-th element is still a bit annoying since a proof that the index is less then n has to be provided.

Instead of using an initial segment of the natural numbers we could use the modulo operation to ensure that the indexing operation over a tuple makes sense. We could for example think the domain of a tuple as the finite cyclic group $\mathbb{Z}/n\mathbb{Z}$ of order n (if n is greater that zero).

As a side-effect we have a cyclic operation defined on the domain of the tuple, that allows, for example, an intuitive definition of rotations.

```
Let zp := zp_group lt0n.
Definition zprot (f : fgraphType zp d) x := fgraph_of_fun (fun y : zp ⇒f (x ∗ y)).
```

Here ∗ is the group operation and x, as shown in the Figure 2.2, is an element of $\mathbb{Z}/n\mathbb{Z}$ describing the rotation.

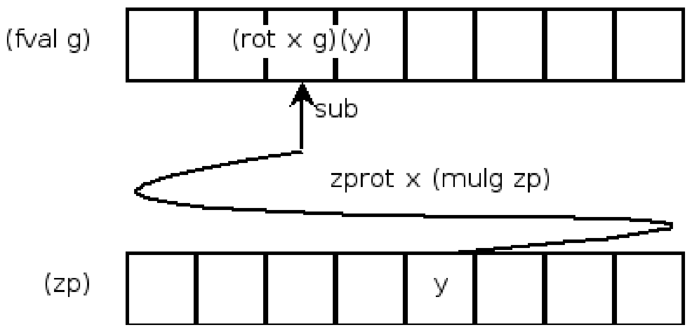


Figure 2.2: Rotated tuple

By the group laws of $*$ we clearly have that (x $*$ y) is inside $\mathbb{Z}/n\mathbb{Z}$, thus indexing f with it makes sense. We also have for free that the unit of the group performs no rotation at all and that the rotation operation is associative. A coercions from natural numbers to elements of the $\mathbb{Z}/n\mathbb{Z}$ group can easily be declared using the modulo operation properties, making indexing and rotations of tuples really easy to write.

2.2.6 Actions

The group action concept is extremely recurrent in group theory, see for example the Cauchy theorem of section 2.2.7.

The usual definition of group action is

Definition 2.1 ((right) group action) *Given a group G and a set X, a (right) group action if a function $f : X \to G \to X$ that validates the following equations:*

- $f(x, (g * h)) = f(f(x, g), h)$
- $\lambda x.f(x, g)$ *is bijective for every g*

where $x \in X$, $g \in G$, $h \in G$, 1 is the unit of the group G and $*$ is the group operation.

The natural representation of such function is what follows

```
Variable (G : finGroupType) (S : finType).
Record action : Type := Action {
  act_f :> S →G →S;
  act_bij : bijective act_f;
 act_morph : ∀(x y : G) z, act_f z (x ∗ y) = act_f (act_f z x) y
}.
```

This definition, although correct, hides an important problem. Actions are object mathematicians usually build on the fly during proofs. This suggests that they must be easy to define. To ease the definition of actions we choose another record type, that is equivalent to the previous one, but requires less effort to be inhabited. It is a well known result that the bijection property can be proved from the fact that the function respects the unit and is a morphism. Thus the following definition is equivalent to the previous one.

```
Record action : Type := Action {
  act_f :> S →G →S;
 act_1 : ∀x, act_f x 1 = x;
 act_morph : ∀(x y : G) z, act_f z (x ∗ y) = act_f (act_f z x) y
}.
```

As an example we prove that the function zprot define in Section 2.2.5 is an action of $\mathbb{Z}/p\mathbb{Z}$ over the set of tuples of length p. Here g2f is an alias for s2f, while gsimpl is a custom tactic defined with Ltac [36] that performs automatic rewriting using some basic equalities valid in any group.

```
Lemma zprot_to1 : ∀f, zprot f 1 = f.
Proof. by move⇒ f; apply/fgraphP⇒ i; rewrite /zprot g2f mul1g. Qed.
Lemma zprot_to_morph : ∀x y f, zprot f (x ∗ y) = zprot (zprot f x) y.
Proof. move⇒ x y f; apply/fgraphP⇒ i; rewrite /zprot !g2f; gsimpl. Qed.
Canonical Structure zprot_action := Action zprot_to1 zprot_to_morph.
```

We again use the canonical structure mechanism to attach to the function zprot its action properties as we did for groups (see Section 2.2.1). This allows Coq to infer these properties when needed.

2.2.7 A proof in SSReflect

The aim of this section is to show a "real" proof in SSReflect. So far, only two lines long lemmas where proved. Here we present the proof of a statement that actually has a name, the Cauchy theorem. Not all lemmas and definition used in the theorem have been describe right now, and many of them will be explained informally when needed.

We begin reporting the statement of the Cauchy theorem as well as the proof that can be found on Planet Math[1].

Theorem 2.1 (Cauchy) *Let H be a finite group and let p be a prime dividing $|H|$. Then there is an element of H of order p.*

Proof: Let H be a finite group, and suppose p is a prime divisor of $|H|$. Consider the set X of all p-tuples $(x_1, \ldots, x_p)$ for which $x_1 \cdots x_p = 1$. Note that $|X| = |H|^{p-1}$ is a multiple of p. There is a natural group action of the cyclic group $\mathbb{Z}/p\mathbb{Z}$ on X under which $m \in \mathbb{Z}/p\mathbb{Z}$ sends the tuple $(x_1, \ldots, x_p)$ to $(x_{m+1}, \ldots, x_p, x_1, \ldots, x_m)$. By the Orbit-Stabilizer Theorem, each orbit contains exactly 1 or p tuples. Since $(1, \ldots, 1)$ has an orbit of cardinality 1, and the orbits partition X, the cardinality of which is divisible by p, there must exist at least one other tuple $(x_1, \ldots, x_p)$ which is left fixed by every element of $\mathbb{Z}/p\mathbb{Z}$. For this tuple we have $x_1 = \ldots = x_p$, and so $x_1^p = x_1 \cdots x_p = 1$, and x_1 is therefore an element of order p. $\square$

The hypothesis of the theorem, as well as some constructions, will be declared as variables or local definitions. The context in which the theorem is proved is shown in Figure 2.3.

Most hypothesis and local definitions are one to one with the pen and paper proof. zp1 is the unit of the zp group domain. The function prod_over_zp takes

[1]`http://planetmath.org/?op=getobj&from=objects&id=1569`

```
elt : finGroupType
H : group elt
p : nat
prime_p : prime p
p_divides_H : dvdn p (card H)
lt1p := prime_gt1 (p:=p) prime_p : 1 < p
zp := zp.zp_group (p:=p) (ltnW (m:=1) (n:=p) lt1p) : finGroupType
zp1 := make_zp 1 : zp
prod_over_zp (f : zp →elt) :=
  foldr (fun (i : nat_eqType) (x : elt) ⇒f (zp1 ** i) * x) 1 (iota 0 p))
X := {t : fgraph_finType zp elt, tfunspace (d2:=elt) H t && (prod_over_zp t == 1)}
```

Figure 2.3: Context of Cauchy theorem.

in input a function of type zp →elt (possibly a tuple over elt with p elements) and computes $\Pi_{i=0}^{p-1} f_i$. The operation ** is exponentiation on group elements (and zp1 ** (n−1) gives the n-th element of zp). X is the set of tuples (whose elements lay in H) such that the product of their element is 1. The lemma card_X, proved previously, states that card X = card H ˆ (p−1).

The whole proof script is reported in Figure 2.4 while a detailed explanation of every line follows.

Line 3 proves an additional lemma that will be put in the context: the cardinality of zp_group is p. zp_group, defined before, is an object of type group zp using the whole group domain zp (i.e. the biggest group living in zp).

Line 4 uses the so called "mod p lemma" to show that modn (card X) p equals modn (card Z) p where Z is set to (act_fix zprot_action zp_group)∩X, where modn is modulus operation over natural numbers. The "mod p lemma" states that, given a group H that acts with f on a finite set A, and a prime number p such that the cardinality of H equals pˆn for some n, then the cardinality of modn (card A) p equals to modn (card A∩(act_fix to H)) p, where act_fix to H is the set of points of A invariants by the action f.

```
1  Theorem cauchy: ∃a, H a && (card (cyclic a) == p).
2  Proof.
3  have card_zp: card zp_group = p ^ 1 by rewrite icard_card card_ordinal /= muln1.
4  have:= mpl prime_p card_zp zprot_acts_on_X; set Z := setI _ _.
5  rewrite card_X −{1}(leq_add_sub lt1p) /= −modn_mul (eqnP p_divides_H) /=.
6  pose t1 := fgraph_of_fun (fun _ : zp ⇒1 : elt ).
7  have Zt1: Z t1.
8    apply/andP; split; [| rewrite s2f; apply/andP; split].
9      by apply/act_fixP ⇒ x _; apply/fgraphP ⇒ i; rewrite /t1 !g2f.
10     by apply/tfunspaceP ⇒ u; rewrite g2f.
11   rewrite /prod_over_zp; apply/eqP; elim: (iota _ _) ⇒//= i s → {s}.
12   by rewrite g2f mul1g.
13 case: (pickP (setD1 Z t1)) ⇒[t | Z0]; last first .
14   by rewrite mod0n (cardD1 t1) Zt1 (eq_card0 Z0) modn_small.
15 case/and3P ⇒ tnot1; move/act_fixP ⇒ fixt.
16 rewrite s2f; case/andP; move/tfunspaceP ⇒ Ht prodt _.
17 pose x := t 1; ∃x; rewrite Ht /=.
18 have Dt: t _ = x by move ⇒ u; rewrite /x −{2}(fixt u (isetAP _)) g2f mulg1.
19 have: dvdn (orderg x) p.
20   rewrite orderg_dvd −(eqP prodt) −(size_iota 0 p) /prod_over_zp.
21   by apply/eqP; elim: (iota _ _) ⇒//= i s ←; rewrite Dt.
22 case/primeP: prime_p ⇒_ divp; move/divp; case/orP; rewrite eq_sym //.
23 move/eqP ⇒ Dx1; case/eqP: tnot1; apply/fgraphP ⇒ i.
24 by rewrite Dt −(gexpn1 x) /t1 g2f −Dx1 (eqP (orderg_expn1 _)).
25 Qed.
```

Figure 2.4: Cauchy theorem proof.

Line 5 simplifies the previously introduced hypothesis to

```
modn (card H ^ (p − 1)) p = modn (card Z) p
```

Then using the hypothesis lt1p the subterm (card H ^ (p −1)) can be rewritten to

(card H ∗ card H ˆ (p − 2)). Since p divides card H (by assumption p_divides_H) we can conclude that (modn 0 p = modn (card Z) p).
Line 6 defines t1 as the p-tuple $(1, \dots, 1)$.

Line 7 asserts that t1 is in Z. In line 8 the boolean intersection predicate is reflected to the propositional conjunction, then the goal is split in two parts:
t1∈act_fix zprot_action zp_group
and t1∈X. The second goal is suddenly simplified exposing the definition of X (rewriting with s2f) and the resulting boolean conjunction is split after being reflected to the propositional one.
Line 9 uses the view lemma act_fixP to switch from the boolean predicate act_fix to its propositional counterpart. Given an action a of a group G over a set T, we say that $t \in T$ is fixed by a if for every $x \in G$ we have that $a(t, x) = t$. After the view application, the goal is changed to

```
∀x : zp, zp_group x →zprot_action t1 x = t1
```

Then x is introduced and the hypothesis x∈zp_group is discharged with the bookkeeping command ⇒x _. Since t1 is a tuple, that is also a function represented using the function graph construction we built in Section 2.2.3, the extensionality rule for functions, proved with fgraphP, holds and can be used as a view. Introducing i we are left to prove zprot_action t1 x i = t1 i. Then t1 is unfolded. Here we also unfold zprot_action and zprot to ease the understanding of the following step, but it not strictly needed since these unfoldings are done automatically by SSReflect if needed.

```
fgraph_of_fun (fun y : ordinal p ⇒fgraph_of_fun (fun _ : ordinal p ⇒1) (x ∗ y)) i =
  fgraph_of_fun (fun _ : ordinal p ⇒1) i
```

Rewriting with g2f multiple times and reducing β-redexes leads to a trivial goal $1 = 1$.
Line 10 proves that t1 belong to the total function space whose codomain is H (actually a subset of elt). Since t1 is only composed by units, and since H is a group, H (t1 u) is always true for every u in zp.

Lines 11 and 12 prove that the product over t1 is 1. It is proved by induction over iota 0 p, the sequences of indices that are used to index t1 in prod_over_zp.

Line 13 picks an element in Z different from t1 (actually picks an element from the set Z {: t1}. The standard pickP predicate can be inhabited in two ways, one that tells you that the set was empty and gives you no element, and the other that gives you an element and proof that this element belongs to the set. The empty set case is processed first.

Line 14 with the first three rewriting steps, reduces the goal to

```
0 = modn (1 + card (setD1 Z t1)) p →∃a : elt, H a && (card (cyclic (G:=elt) a) == p)
```

The following step rewrites card (setD1 Z t1) to 0. Since 1 < p, the assumption is false and the goal solved.

Line 15, using the view and3P on the new assumption that t is in the set setD1 Z t1, obtains that: t1 != t, t is fixed by the action and t belongs to X. The first one is introduced with name tnot1 while the second is introduced as fixt after being reflected to its propositional counterpart with act_fixP.

Line 16 goes further in manipulating the hypothesis on t, obtaining that t belongs to the total function space built with H (Ht : ∀x : zp, H (t x)) and that the product of elements in t is 1 (called prodt). It also drops the hypothesis modn 0 p = modn (card Z) p.

Line 17 exhibits the witness of the existential statement that has to be proved.

```
H (t 1) && (card (cyclic (t 1)) == p)
```

Rewriting with Ht also solves the first part of the conjunction.

Line 18 proves that every element of t is equal to (t 1) (fact called Dt). Rewriting with fixt and g2f leas to a goal t u = t (u ∗ 1) for u in zp.

Line 19 claims that p divides the order of (t 1). The order of (t 1) is the smallest positive natural number n such that (t 1) ˆ n = 1.

Line 20 massages the previous claim, reducing it to

```
(t 1) ** size (iota 0 p) == prod_over_zp t
```

Line 21 proves by induction over (iota 0 p) that (prod_over_zp t) and ** give the same result.

Line 22 reasons on the new hypothesis (dvdn (order (t 1)) p): since p is prime its set of divisors is {1, p}. We thus have two cases, the trivial one is solved in this line and is where (order (t 1)) = p that is exactly what we want to prove (i.e. that the cardinality of the cyclic group over (t 1) is p).
Line 23 and 24 cope with the absurd case, in which (order (t 1)) = 1, thus (t 1) = 1 and by Dt all elements in t are the same. This is in contrast with tnot1.

2.2.8 Considerations on the development

At the time of this writing the fragment of group theory formalized by the Mathematical Components team is already one of the most complete formalisation of finite group theory. It almost covers the material that can be found in an introductory course on group theory. Very few standard results like the simplicity of the alternating group are still missing.

The size of development amounts to 15000 lines of SSReflect vernacular that generate more than 2000 distinct definitions and theorems. Only one third of the total amount of lines is strictly related to group theory, the rest can be considered part of the SSReflect base library, even if many notions like finite sets or homogeneous tuples have been added to ease the development of finite group theory. More than 2000 lines are for the basic definitions and lemmas involving types equipped with a decidable equality, finite types and sequences over these types.

2.3 Methodologies and Issues

During our internship we do not only developed definitions and theorems, but we also put big efforts in maintaining the existing library of theorems. As any software, Coq and SSReflect did not always fulfil our needs and we had to developed workarounds. Moreover some best practices used in proof script development and maintenance have been used and will be discussed.

In this section we present some considerations concerning the tools and the methodologies we used. We also discuss some workarounds we developed. In particular Section 2.3.1 develops a workaround for the type class matching algorithm implemented in Coq that badly interacts with δ-reductions. In Section 2.3.2 we make some consideration on the maintenance of a library of already proved theorems. Section 2.3.3 is devoted to some additional consideration on the interface the system offers to the library of known facts. In Section 2.3.4 we describe an issue of the current Coq tactic language that makes working with SSReflect much more painful that what it could be, namely the execution of tacticals in a big step fashion.

2.3.1 Control of coercions through type tagging

A coercion is a function, which is automatically applied to a term in order to create a new term of the expected type when needed. By default, these functions are not displayed to the user, hence this mechanism can model a notion of subtyping.

Coercions will be discussed in details in Chapter 5, what we do here is to develop a scalable workaround for an issue of standard Coq. Fixing the system can not be considered a possible solution, since it would have an impact on type inference performances. Moreover, SSReflect is a self contained patch (just one .ml4 file) that can be easily linked with standard Coq; changing the type inference algorithm can not be done in the same smooth way.

Before introducing the problem a short introduction on the implementation of coercions in Coq done by Saibi [84] is necessary. In the literature (see for example [11]) coercions domain and codomain is usually computed up to conversion, that is a coercion from A to B is used to explicitly cast an object whose type A' lives in the same equivalence class, determined by β-reduction, of A. Reduction is an expensive operation, and even if Coq has an efficient reduction algorithm for closed terms [47], some restrictions have to be employed to make the problem computationally tractable. Saibi implements coercions between type classes, dropping δ-reduction (i.e. constant expansion) and looking only at the head constant of types.

We now show that some care is needed in combining coercions with structures to avoid unexpected behaviours provoked by reduction. The following toy example illustrates this issue. We pile two types. LowStruct defines the type of lists of natural numbers. The record type HiStructDef compounds a Type with a dummy proof, and its first field sort : HiStructDef →Type is declared as a coercion. The term HiStruct, built with LowStruct, and the canonical proof I of the True proposition, is of type HiStructDef.

```
Definition LowStruct := seq nat_eqType.
Structure HiStructDef : Type := Hi { sort :> Type; prop : True }.
Definition HiStruct := Hi LowStruct I.
Definition sl : LowStruct := Seq 1 2 3.
Definition sh : HiStruct := sl.
```

The term sl has type (convertible to) sort HiStruct, and since sort is a coercion, the above definition of sh is accepted by the system. We then define another coercion, this time from LowStruct to nat :

```
Coercion sum (s : LowStruct) := foldr addn 0 s.
```

Now, (1 + sl) is well typed and evaluates to 7, as well as (1 + sum sh) and (1 + (sh : LowStruct)), but (1 + sh) is rejected by the system. The type of sh is expected to be nat. Since it is not, Coq tries to insert an available coercion. The source of the coercion should match syntactically the type sort HiStruct of sh. If no coercion is found, then the system computes the head normal form of the type and tries again to find a suitable coercion. But "**Eval** hnf **in** (sort HiStruct)" computes the value seq nat_eqType, which does not fit any registered coercion source either. Yet sort HiStruct and LowStruct are convertible. In our development we make an extensive use of coercions, all modularity would be lost without the possibility of piling structures.

This over-reduction phenomenon is inhibited by a general type-tagging method, which consists in wrapping the lower structure in a single constructor coinductive type.

```
CoInductive LowStruct : Type := Tag : seq nat_eqType →LowStruct.
Definition untag (l : LowStruct) := match l with Tag s ⇒s end.
```

Note that untag, the projection associated to the wrapper, is *not* declared as a coercion, thus a term of type LowStruct can not be coerced (nor reduced) to a term of type nat_eqType.

It is possible to prove once and for all the three lemmas allowing to build generically the tagged fin/eqType versions of an untagged fin/eqType. We need a tag constructor, and an untag projection operator. These satisfy a (cancel untag tag) condition, namely: ∀x, tag (untag x) = x.

First we provide an eqP lemma to build the eqType structure:

```
Lemma can_eq:
  ∀(d : eqType) (tag_d : Type),
  ∀(tag : d →tag_d) (untag : tag_d →d),
   cancel untag tag →
     reflect_eq (fun (x y : tag_d) ⇒untag x == untag y).
```

Then we provide the enumP property required for building the finType. The undup operator removes possible duplicates created by the mapping of the tag function over the elements of the initial finType, making the proof of the can_uniq lemma simpler.

```
Definition
  uniqmap (d1 : finType) (d2 : eqType) (tag : d1 →d2)
:=
  undup (maps tag (enum d1)).
Lemma can_uniq:
  ∀(d1 : finType) (d2 : eqType),
  ∀(tag : d1 →d2) (untag : d2 →d1), cancel untag tag →
    ∀u : d2, count (set1 u) (uniqmap tag) = 1.
```

An example of tagged type is setType (see Section 2.2.3).

2.3.2 Proof maintenance and automation

During our internship in the Mathematical Components team, automation has never been used. The main reason was that it badly interacts with maintenance of a large development. The issue is pretty clear, every form of automation responds differently to changes in the definitions, and is usually hard to understand why a black box complex tool like an automatic procedure fails. Moreover definitions have to be carefully chosen to make the whole huge development run smoothly. Adding to that already difficult choice the additional requirement that they must be automation aware is probably too much.

When definitions are changed proofs break. SSReflect tries to make proof fail as soon as possible, allowing the user to spot the exact step that does not work anymore. It can be a rewriting that does not rewrite anymore, or an application that fails, but in any case the step that used to work is clear. Unless the proof is completely dependent on the definition that changed, a small patch can usually be inserted quickly. Automation hides the step that, during the previous run, lead successfully to a proof. Thus, when it fails the user has no context and no intuition on how to fix the proof script. Sometimes the new definition does not even fit the automatic procedure domain.

We believe that automation has first to be considered in a wider way, as a general assisting tool integrated with searching technology, and then be implemented in a way that gives back to the user the maximum amount of informations possible. In our opinion an automatic tactic must give back to the user a, possibly nice, proof script. That script will offer many benefits: it will execute faster than the automatic tactic the next time the proof script is checked; if it breaks the user may find a quick patch for it; it may use a lemma the searching facility found but that user was not aware of (maybe because it was added to the library in the wrong place or with a meaningless name).

Many users like the compactness of an **auto** command, but we believe this is a user interface issue. Editors for programming language support folding functions definitions for example, expanding them when requested. The same could be done here, allowing the editor to collapse the proof script generated by the automatic tactic if the user wants so. Another common critic to that approach is that some user like that fact that automatic tactic sometime succeed in finding a proof even if some definitions or lemma changed, but explicit proofs always break. We believe that this issue regards only the way the proof script is generated by the automatic tactic. The proof script may be generated in such a way that if it fails the automatic tactic is run (with the first tactical for example). As we will see in Chapter 6 MATITA has a syntactic distinction between commands that behave interactively and may modify the script file and commands that can not, like **auto** and autobatch. The former outputs a proof script, that may use autobatch (possibly tuning it with the a-posteriori knowledge of the proof found) as a fallback.

We also think that automation and searching facilities has to assist the user all the time. A simple example is the duplicate check MATITA performs every time the user starts the proof of a lemma. A duplicate may be found both looking for a similar lemma using searching facilities and using automatic tactics with a very limited time or step limit. If **auto** is able to prove a lemma in two steps (maybe using lemmas in the library the user is not aware of), the user may like to be warned that the result she wants to prove is not that interesting (if placed in the context where **auto** succeeded in proving it). The same kind of check may be done when re-executing a proof script. New lemmas added before a given result may make it really useless or oversimplify its proof. This actually happened when many parts of the MATITA standard library where tried to be proved by means of automation, revealing that some lemmas were obsolete. Moreover many of these check, that are actually time consuming, may be performed only in interactive mode, and could be switched on/off in response to a user request.

2.3.3 Naming and searching lemmas

When we joined the Mathematical Components group, we had no previous experience about using Coq. Everybody agrees that formal proofs are tricky, but in our experience what really makes you feel lost is the library of available results.

The interface to the prover used by the Mathematical Components team is Proof General [5] that, at the time we used it, was missing some feature we consider important. Both the Matita user interface and CoqIde give a quick idiom (single click the former; select and click the latter) to reach and display the definition of an object. The Print and Check commands badly interact with modules and implicit argument, forcing the user to seldom try combinations of @ and _._ before having the definition displayed.

The search command implemented by Georges Gonthier during our internship has already been described in 2.1.1. Although this search facility is performed only on modules (set of lemmas) that are explicitly required, it has shown to be extremely useful. The overall design of Matita, that will be better analysed in Chapter 4, allows to search the whole library and not only the loaded part. This can result in a benefit if the library is huge and composed by modules the user does not see and thus can not guess are worth being required. Moreover, the user usually prefers a lemma that can be applied or an equation that can actually rewrite something. Here, our wide view on automation, comes again to play. An automatic rewriting tactic can be used, with a very small amount of time limit, to check if a lemma returned by the searching facility can rewrite the goal, and rate it high if it can. The same holds for applicative reasoning, and, for example, is already implemented in the hint Matita command, that gives only lemmas that can be actually applied successfully to the current goal.

Even if the user is assisted by searching facilities, a good naming policy for lemmas ease the process of remembering (or even guessing) lemmas names. SSReflect follows a good practice in the whole base library, that is not enforced by the system. This results in extra modules not being sticky to that policy, especially when written by people already having their own taste for lemmas names. We found the naming schema really handy to use, and in some aspects even better then the one that is implemented in Matita, that in our opinion suffers from enforcing long, clearly meaningful, names. Matita has an hard-coded naming policy, enforced automatically by the system that warns the user when the chosen name does not fit the policy. We believe that the support of the system in spotting bad names is essential, thus the policy has to be enforced by the system. For simplicity Matita has an hard-coded policy described in [3], but a way to customise the policy should really be implemented.

2.3.4 Complex commands and big step execution

The SSReflect extension allows to build complex commands out of simple operations. Anyway, predicting the result of the execution of many commands at a time is hard, thus these complex commands are built up piece by piece. Because of the interaction model imposed by Proof-General to extend an already executed command with an additional step the user needs to backtrack one step, edit the command and re-execute it. This annoying behaviour tends to make the user learn how to better predict the system output and increases her ability to write more complex commands at once. Although this side effect can be considered positive after all, the same behaviour is extremely painful when proofs are re-executed. Note that a proof really has to be executed to understand what is going on, and that this happen not only during demos or talks, but every time it breaks.

SSReflect commands are built on top of primitive ones, composing them using Coq tactic language Ltac [36]. Ltac expressions are evaluated in a big step fashion, making it impossible for the user interface to show intermediate steps. Moreover, this behaviour has a general unpleasant impact on how proofs are structured in standard Coq (see Section 4.3.2), usually resulting in completely unstructured proof scripts. At the beginning of our PHD we developed a small step semantic for a subset of LCF tacticals [81] that will be described in Chapter 4. The small step approach solves the problems presented above, but needs some modifications to the tactic language execution engine and parser to be implemented.

Chapter 3

D.A.M.A.

During the last part of out PHD we contributed to the D.A.M.A. Project[1], leaded by Sacerdoti Coen and funded by the University of Bologna. The project has three major goals:

- improvement and specialisation of the interactive theorem prover Matita
- development on top of Matita of a learning environment for students to verify their advancement in doing mathematical proofs
- formalisation of abstract measure theory in Matita up to the Lebesgue's Dominated Convergence Theorem

We worked on the first and the last items. In this chapter we describe the formalisation we made pointing out which features of Matita allowed such formalisation. These features will be analysed in details in the second part of the manuscript (Chapter 5) where their implementation will also be described.

The formalisation work described in this chapter has not been published yet, mainly because we obtained that result in the very last part of our PHD. Moreover, we built all the base infrastructure of lemmas to attack Lebesgue dominated convergence theorem, without proving it. We stopped, because of lack of time, at the well known sandwich theorem, that proved to be a reasonable test case for our framework and a cornerstone for the future formalisation of the dominated convergence theorem.

3.1 Lebesgue dominated convergence theorem

Let $f_1, f_2, f_3, \ldots$ denote a sequence of real-valued measurable functions on a measure space (S, Σ, μ). Assume that the sequence converges pointwise and is dominated by some integrable function g. Then the pointwise limit is an integrable function and

$$\int_S \lim_{n\to\infty} f_n \, d\mu = \lim_{n\to\infty} \int_S f_n \, d\mu.$$

[1] `http://dama.cs.unibo.it/`

To say that the sequence is "dominated" by g means that

$$|f_n(x)| \leq g(x)$$

for all natural numbers n and all points x in S. By integrable we mean

$$\int_S |g| \, d\mu < \infty$$

A σ-algebra over a set S is a nonempty collection Σ of subsets of S that is closed under complementation and countable unions of its members. A measure space (S, Σ, μ) contains a σ-algebra Σ over S and a function $\mu : \Sigma \longrightarrow \mathbb{R}$.

This is the classic formulation of the theorem and its proof can be found in many textbooks like the one by Rudin [79].

A more general formulation, see for example Fremlin [38], drops the σ-algebra structure in favour of the study of more topological/algebraic structures like Riesz spaces, also called vector lattice. These algebraic structures are ordered vector spaces equipped with a norm where the ordered set forms a lattice. In this setting it is relevant to study the convergence in terms of the order relation induced by the lattice operations and in terms of the norm. Integration is thus an example of a linear function from elements of the vector lattices to $\mathbb{R}$.

An even more general, and still classic, approach is the one of Weber [50, 51] that drops the vectorial lattice in favour of a metric lattice, and compares the convergence in the sense of order relation given by the lattice and the convergence in the sense of the metric.

On the constructive side we find works from Bishop and Bridges [17] and Spitters [88] who follows the same approach of Fremlin, but proves only a corollary of the Lebesgue dominated convergence theorem.

3.1.1 The constructive version developed in D.A.M.A.

The theoretical study for the D.A.M.A. Project has been done by Enrico Zoli, who developed a complete constructive proof of the Lebesgue dominated convergence theorem in a setting inspired by Weber. In Figure 3.1 the overall picture is presented.

Some structures like excess and apartness are not widespread notions and in the following paragraph we will only scketch their meaning; the interested reader can find a good introduction, both historical and technical, in [9, 66].

An apartness is a structure equipped with a coreflexive, symmetric and cotransitive relation that, when negated, generates an equivalence relation. An excess is a structure equipped with a coreflexive and cotransitive relation that, when negated, generates a partial order relation. Both relations, when used as primitive, subtend a computational meaning: for example the apartness relation expresses how far two objects are. Its negation hides that computational content, making the derived equality notion suitable for settings in which equality between two objects is in general undecidable (and thus must subtend no computational meaning). The most relevant example is the set $\mathbb{R}$ of real numbers, that can be represented possibly infinite sequences of digits. No procedure can state if two real numbers (essentially functions from $\mathbb{N}$ to $\{0, \ldots, 9\}$) are equal in a finitary time. On the contrary there exists a procedure to semi-decide if two real numebrs differ, and this procedure can also calculate their distance (for example the position of the first different digit).

In Figure 3.1 every box represents an algebraic structure, every double arrow represents inheritance. Regular arrows represents coercions not used to mimic subtyping but stating that a structure is a models of another one, for example a lattice induces an excess relation defining $a \nleq b \equiv a \# a \wedge b$. The semi metric space structure, as well as the pre weighted lattice, is parametric over a totally ordered divisible group; we omit additional arrows representing these dependencies for the sake of readability. Inside boxes we report only the types and the operations defined over them each structure embeds. We omit the properties these operations must validate, again for the sake of readability.

The constructive proof developed by Zoli lives in the semi metric lattice structure, but he also exhibits two models that are represented with boxes with a double border.

In the next section we will describe our formalisation in the Matita interactive theorem prover of all the single border boxes, and we use these structures to state and prove the sandwich theorem, a cornerstone of the dominated convergence theorem.

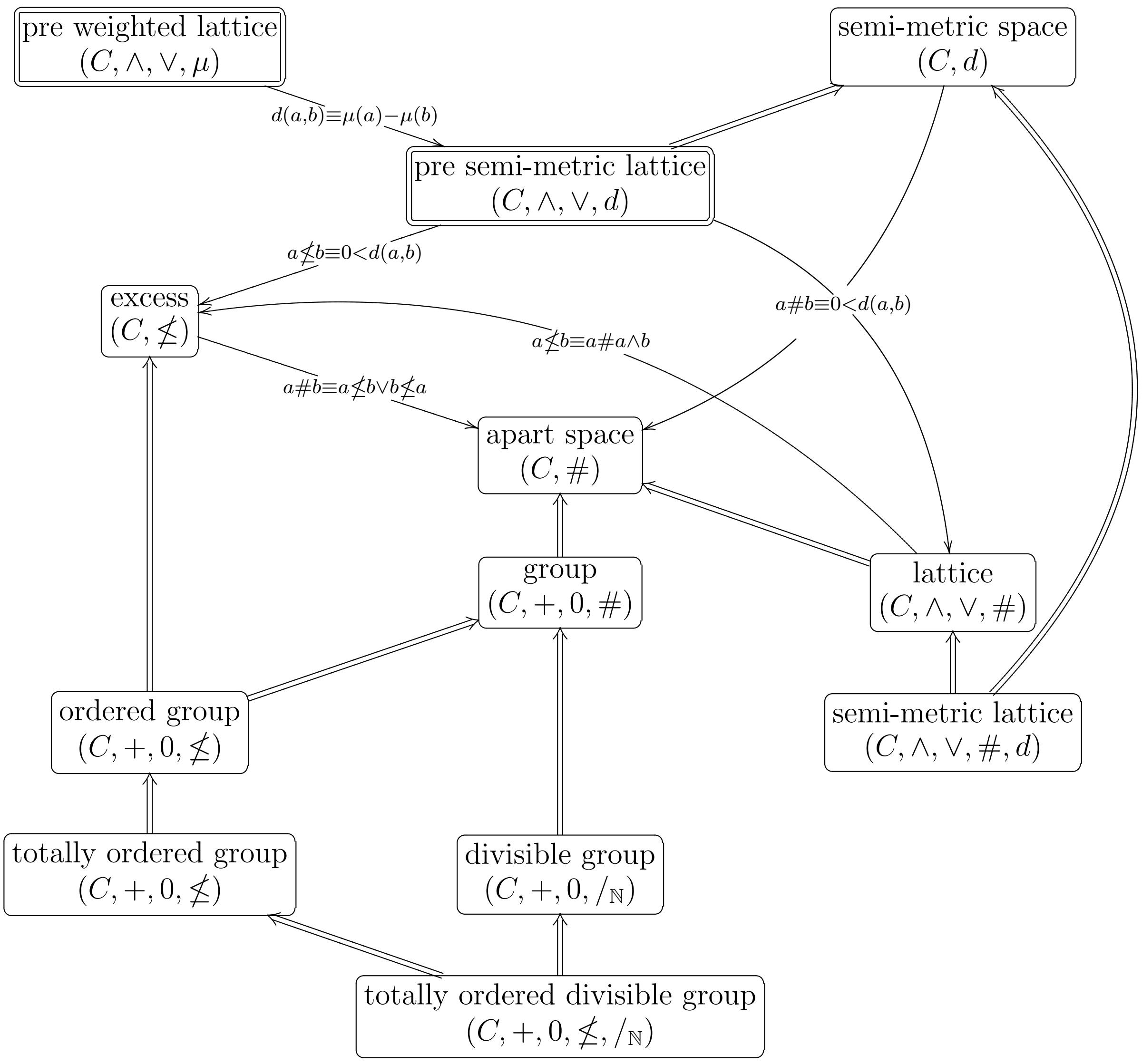

pre weighted lattice $(C, \wedge, \vee, \mu)$
semi-metric space (C, d)
$d(a,b) \equiv \mu(a) - \mu(b)$
pre semi-metric lattice $(C, \wedge, \vee, d)$
$a \not\leq b \equiv 0 < d(a,b)$
excess $(C, \not\leq)$
$a \# b \equiv 0 < d(a,b)$
$a \not\leq b \equiv a \# a \wedge b$
$a \# b \equiv a \not\leq b \vee b \not\leq a$
apart space $(C, \#)$
group $(C, +, 0, \#)$
lattice $(C, \wedge, \vee, \#)$
ordered group $(C, +, 0, \not\leq)$
semi-metric lattice $(C, \wedge, \vee, \#, d)$
totally ordered group $(C, +, 0, \not\leq)$
divisible group $(C, +, 0, /_{\mathbb{N}})$
totally ordered divisible group $(C, +, 0, \not\leq, /_{\mathbb{N}})$

Figure 3.1: Inheritance graph in DAMA

3.2 The formalisation in Matita

In the last months of our PHD we took up the formalisation of this topic. Although previous attempts of formalisation were there, mainly concerning the classical version and one of its variants, we started from scratch. These formalisations were sketched, where Leibniz equality was used instead of a weaker one to speed up development, but dropping all interesting model of the formalised results. Moreover, these formalisations never reached a reasonable status, since they were worked on when Matita was still unable to handle algebraic structures in a proper way. From the experiences that came from these attempts we developed the technique we adopted in the development, whose system-side counterpart is detailed in Chapter 5.

Another consideration, regarding setoids, has to be done. Since we work with objects whose equality can not be assumed to be decidable, rewriting has to be performed with weaker principles than the one validated by Leibniz equality. Sacerdoti developed in Coq the tactic Setoid Replace [25] to allow the user to perform rewriting is such weaker framework with great simplicity. The tactic automatically composes lemmas about morphisms to perform deep rewritings, an extremely tedious and uninteresting task. The porting of that tactic to Matita has not ended yet, but we wanted to build the basement for an interesting development, thus we dropped the use of Leibniz equality. To overcome most of the tediousness of deep rewriting steps we used automation and some flavour of coercions from and to the same type we will detail later.

Last, Matita offers notational support and we used it constantly, but lacks automatic calculation of implicit arguments. In the proof scripts that will follow many question marks will appear in place of arguments the system is able to infer.

The clear organisation of structures given in Figure 3.1 draws a clean path we followed in the formalisation.

3.2.1 The excess relation

The first building block is the definition of the excess relation, the apartness relation and the coercion from an excess relation to an apartness relation.

```
record excess : Type :={
  exc_carr:> Type;
  exc_relation : exc_carr → exc_carr → Type;
  exc_coreflexive : coreflexive ? exc_relation ;
  exc_cotransitive : cotransitive ? exc_relation
}.
```

The cotransitive property is defined as follows using the higher order facilities CIC offers:

```
definition cotransitive :=λ C:Type.λ lt:C→ C→ Type.∀ x,y,z:C. lt x y → lt x z ∨ lt z y.
```

The excess relation, having a computational content, is used to define the usual less or equal relation, not having a computational content. The $\leq$ relation is then proved to be a partial order relation. Note that we will use $\nleq$ for the excess relation.

```
definition le :=λ E:excess.λ a,b:E. ¬ (a ≰ b).
lemma le_reflexive: ∀ E.reflexive ? (le E).
lemma le_transitive: ∀ E.transitive ? (le E).
```

The less or equal relation still misses the antisymmetric property, since to state it we need to define the apartness relation and its negation, the equality. We will use # for the apartness relation and $\approx$ for its negation.

```
record apartness : Type :={
  ap_carr:> Type;
  ap_apart: ap_carr → ap_carr → Type;
  ap_coreflexive : coreflexive ? ap_apart;
  ap_symmetric: symmetric ? ap_apart;
  ap_cotransitive : cotransitive ? ap_apart
}.
definition eq :=λ A:apartness.λ a,b:A. ¬ (a # b).
lemma le_le_eq: ∀ E:excess.∀ a,b:E. a ≤ b → b ≤ a → a ≈ b.
lemma eq_le_le: ∀ E:excess.∀ a,b:E. a ≈ b → a ≤ b ∧ b ≤ a.
lemma le_antisymmetric: ∀ E:excess.antisymmetric ? (le E) (eq E).
```

To allow the system to accept the latter statement, we need to define the apartness relation induced by the excess relation. This is done in the usual way, saying that a is apart from b if a is is in excess over b or vice versa.

```
definition apart :=λ E:excess.λ a,b:E. a ≰ b ∨ b ≰ a.
definition apart_of_excess : excess → apartness.
coercion cic:/matita/excess/apart_of_excess.con.
```

The ≈ relation can be proved to be an equivalence relation. This gives us the possibility of building chains of equalities. Since we will use the transitivity property almost everywhere, we defined a notational shortcut (mainly to hide many implicit arguments).

```
lemma eq_trans:∀E:apartness.∀x,z,y:E.x ≈ y → y ≈ z → x ≈ z.
notation > "'Eq'≈" non associative with precedence 50 for @{'eqrewrite}.
interpretation "eq_rew" 'eqrewrite = (cic:/matita/excess/eq_trans.con _ _ _).
```

The flavour of proof script obtained using the Eq≈ notation is mostly declarative. Consider the following sequent.

```
H1 : a ≈ b
H2 : c ≈ b
================
a ≈ c
```

The following proof lines are all valid:

```
apply (Eq≈ ? H1);
apply (Eq≈ (a≈b) ? H2);
```

The former command rewrites a into b by means of H1, the second line is much trickier, since it involves the coercion eq_sym. Since Eq≈ is applied to (a≈b) its third argument is expected to be of type b≈c but is of type c≈b. The refiner (type inference subsystem) of Matita tries to fix every ill typed application inserting coercions. As it will be detailed in Chapter 5 Matita allows the user to declare coercion from and to the same type. As in the implementation of coercive subtyping made by Saibi [84] in Coq and by Bailey [7, 8] in Lego, the source and target type of coercion is approximated with the head constant symbol, in that case, eq_sym is register as a coercion from ≈ to ≈. The expected type and the actual argument type approximation match the target and source approximation of the eq_sym coercion, Matita tries to insert it. This trick is extensively used in the development, and around sixteen coercions are declared from and to ≈.

The next step is to declare the strictly less than relation and prove it is coreflexive and transitive.

```
definition lt :=λ E:excess.λ a,b:E. a ≤ b ∧ a # b.
theorem lt_to_excess: ∀ E:excess.∀ a,b:E. (a < b) → (b ≰ a).
```

Then, a bunch of lemmas to support rewriting has to be proved and some notation is associate with it, for example the following snippet proves a lemma to rewrite on the left hand side of the excess relation.

```
lemma exc_rewl: ∀ A:excess.∀ x,z,y:A. x ≈ y → y ≰ z → x ≰ z.
notation > "'Ex'≪" non associative with precedence 50 for @{'excessrewritel}.
interpretation "exc_rewl" 'excessrewritel = (cic:/matita/excess/exc_rewl.con _ _ _).
```

The syntax Op≪ and Op≫ has been declared for every Op in $\nleq$, $\leq$ and # and associated to their rewriting principles. Note that the **interpretation** command declares some arguments as implicit (question marks are added on the fly for them). The first non implicit argument in exc_rewl is y, so that the user can specify the left hand side of $\nleq$ after the rewriting.

Given the excess relation, and in particular the apartness relation, we can build the hierarchy of groups.

3.2.2 The hierarchy of groups

The initial step is to define the group structure. We decided to start directly with abelian groups, without defining groups in their full generality. We use additive notation for the group operation, calling the identity 0. In our constructive setting, the strong extensionality law holds for the operation the group is equipped with. This property states that z + x # z + y → x # y for any x, y and z.

```
record abelian_group : Type :={
   carr:> apartness;
   plus: carr → carr → carr;
   zero: carr;
   opp: carr →carr;
   plus_assoc_: associative ? plus (eq carr);
   plus_comm_: commutative ? plus (eq carr);
   zero_neutral_: left_neutral ? plus zero;
   opp_inverse_: left_inverse ? plus zero opp;
   plus_strong_ext: ∀z.strong_ext ? (plus z)
}.
```

Note that the associative and commutative higher order predicates are abstracted over the equality relation that is given as the last argument. Again, many lemma useful for rewriting have been proved and some of them declared as coercions, for example:

```
lemma feq_plusl: ∀G:abelian_group.∀x,y,z:G. y ≈z →x+y ≈x+z.
```

This lemma allows to perform a rewriting in the context (x +·)

Here we say that the abelian group inherits from the apartness structure, since it embeds an apartness and the coercion carr, automatically declared by the :> syntax, allows to apply every lemma proved in the previous section to a group. For example the same eq_trans lemma can be used to rewrite group elements.

At that point the first structure inheriting from multiple structures can be declared. Ordered groups inherit an excess relation and an abelian group. As a first step we declare a technical structure, embedding both the excess relation and the group, but inheriting only from the excess relation. Our objective is to declare a structure that behaves as a group on whose elements an excess relation is defined. Clearly, simply embedding both structures would have not worked, since the carrier of the excess relation (the type of the objects on which the excess is defined) is distinct from the carrier of the group. This makes it impossible to write formulas like 0 < x since 0 has a type completely unrelated to the one < is defined on.

```
record pogroup_ : Type :={
  og_abelian_group_: abelian_group;
  og_excess:> excess;
  og_with: carr og_abelian_group_ = apart_of_excess og_excess
}.
```

The og_with field states that the apartness derived from the excess relation is equal to the apartness relation the abelian groups embed. Here the = symbol stands for the Leibniz equality. For the sake of simplicity we declared the entire apartness structures to be equal. This constraint could be relaxed to stating that the carrier of both apartness relation is the same.

We then build the coercion that manifests the group structure embedded in the pogroup_ structure. Since the objects of the group must be in the same type the excess relation is defined on, we build a group whose carrier is not the one embedded in og_abelian_group_ but is the apartness relation derived from og_excess.

```
lemma og_abelian_group: pogroup_ → abelian_group.
intro G; apply (mk_abelian_group G); [1,2,3: rewrite < (og_with G)]
[apply (plus (og_abelian_group_ G));|apply zero;|apply opp]
cases (og_with G); simplify;
[apply plus_assoc|apply plus_comm|apply zero_neutral
|apply opp_inverse|apply plus_strong_ext]
qed.
```

This script builds an abelian group with mk_abelian_group using the pogroup_ named G (actually the apartness derived from its excess thanks to the apart_of_excess coercion) as its apartness. Then it rewrites with the og_with assumption the first three goals. The syntax 1,2,3: is part of Matita tactic language, and groups together some branches of a [tactical. The full syntax and semantic of Matita tacticals is described in Section 4.3.2.

These first three goals are the constants of the group, namely addition, identity and inverse. Their type before the rewriting step was involving the carrier of the group we are building, thus the apartness derived from the excess of G. After the rewriting, their type involves only the carrier of the og_abelian_group_ structure, and thus they can be inhabited with the constants of such structure. The remaining goals are the properties of the group, but can not be directly proved using the properties of the group embedded in G (namely og_abelian_group_) since, in their type, appears the rewriting made to close the first three goals. For example, when proving the associativity of the group operation, instead of plus the user faces the following term:

```
eq_rect apartness (carr (og_abelian_group_ G))
 (λ a:apartness.ap_carr a→ ap_carr a→ ap_carr a) (plus (og_abelian_group_ G))
 ( apart_of_excess (og_excess G)) (og_with G)
```

The **cases** (og_with G) command performs pattern matching on the og_with field of G on every remaining goal. That behaves again as a rewriting step that also replaces all occurrences of (og_with G) with the constructor of the equality eq_refl that makes all eq_rect applications compute away. After that, all properties can be proved using the properties of the embedded group og_abelian_group_.

Notice that, all steps involving a pattern match on (og_with G) have been delayed as long as possible. In this way the coercion builds a group structure whose first four fields can be projected and actually compute to their content since reduction is not blocked by a pattern match on a possibly abstracted term like og_with. More to the point, the system is now able to accept formulas like the following one:

```
lemma example: ∀ G:pogroup_. ∀ x,y:G. 0≈ x → y≈ x+y.
```

Here the partially ordered group G can be used both as an excess relation (thanks to the og_excess coercion) and as an abelian group thanks to just defined coercion. Moreover, to accept the last part of the statement the return type of plus has to be the same of the variable y (the left hand side of ≈). This is true since the coercion we defined produces a group structure whose carrier *is* the type of y and computes to it. Omitting notation, the last part of the statement is:

```
eq ( apart_of_excess (og_excess G)) y (plus (og_abelian_group G) x y)
```

The return type of plus is thus ap_carr (carr (og_abelian_group G)) that reduces to ap_carr (apart_of_excess (og_excess G)) that is exactly the input type of eq.

The technique we use is explained in details in [27] and in Chapter 5 of this thesis, here we assume the system is able to infer implicit arguments hidden by notation.

We now define the real partially ordered group structure with its strong extensionality property, that could not be declared before since it involves operations of the group and the excess relation.

```
record pogroup : Type :={
  og_carr:> pogroup_;
  plus_cancr_exc: ∀f,g,h:og_carr. f+h ≰ g+h → f ≰ g
}.
```

Then some interesting results are proved like the following lemma that is used almost everywhere in the development.

```
lemma lt0plus_orlt: ∀G:pogroup. ∀x,y:G. 0 ≤ x → 0 ≤ y → 0 < x + y → 0 < x ∨ 0 < y.
```

To define the structure of totally (or linearly) ordered group is enough to inherit from the partially ordered group structure adding the additional totality property.

```
record togroup : Type :={
  tog_carr:> pogroup;
  tog_total: ∀x,y:tog_carr.x≰ y → y < x
}.
```

Inside totally ordered groups, the following key lemma is valid.

```
lemma eqxxyy_eqxy: ∀G:togroup.∀x,y:G. x + x ≈ y + y → x ≈ y.
```

An ℕ-division operation can be defined starting from exponentiation, that in an additive group is usually written like a non commutative multiplication.

```
let rec gpow (G : abelian_group) (x : G) (n : nat) on n : G :=
  match n with [ O ⇒ 0 | S m ⇒ x + gpow ? x m].
```

Then a divisible group is a group together with the property that for every element x and natural number n there exists a y such that x is equal to S n ∗ y.

This, when the existential is in Type, gives a division operation by a positive natural number.

```
record dgroup : Type :={
  dg_carr:> abelian_group;
  dg_prop: ∀x:dg_carr.∀n:nat.∃y.S n ∗ y ≈ x
}.
```

To define totally ordered and divisible groups the same technique used for partially ordered groups is adopted. Inside this algebraic structure the following lemmas can be proved. The former is of fundamental importance for the sandwich theorem.

```
lemma divide_preserves_lt: ∀G:todgroup.∀e:G.∀n.0<e → 0 < e/n.
lemma muleqplus_lt:
  ∀G:todgroup.∀x,y:G.∀n,m. 0<x → 0<y → S n ∗ x ≈ S (n + S m) ∗ y → y < x.
```

3.2.3 The metric space and lattice hierarchy

The definition of lattices poses no new difficulty. The lattice structure embeds and inherits an apartness relation, defines the meet and join operations and their properties.

```
record lattice : Type :=
  l_carr :> apartness;
  join : l_carr → l_carr → l_carr;
  meet: l_carr → l_carr → l_carr;
  join_refl : ∀x:l.(x ∨ x) ≈ x;
  meet_refl : ∀x:l.(x ∧ x) ≈ x;
  join_comm: commutative ? join (eq l_carr);
  meet_comm: commutative ? meet (eq l_carr);
  join_assoc : associative ? join (eq l_carr );
  meet_assoc: associative ? meet (eq l_carr );
  absorbjm: ∀f,g:l. (f ∨ (f ∧ g)) ≈ f;
  absorbmj: ∀f,g:l. (f ∧ (f ∨ g)) ≈ f;
  strong_extj: ∀x:l.strong_ext ? (λy.x ∨ y);
  strong_extm: ∀x:l.strong_ext ? (λy.x ∧ y)
}.
```

The next step is to equip the lattice with an excess relation induced by the lattice operations. We follow [9] and define the excess relation as follows and prove it validates the coreflexive and cotransitive properties defining a coercion from a lattice structure to an excess relation.

```
definition excl :=λl:lattice.λa,b:l.a # (a ∧ b).
lemma excess_of_lattice: lattice → excess.
```

The next step is to define what semi metric spaces are. We abstract them over the codomain of the metric, requiring it to be at least an abelian divisible and totally ordered group. We denote with δ x y the metric operation since it is impossible in MATITA to define the usual notation $d(x, y)$, because parentheses are reserved.

```
record smetric_space (R : todgroup) : Type := {
  ms_carr :>Type;
  metric: ms_carr → ms_carr → R;
  mpositive: ∀a,b:ms_carr. 0 ≤ δ a b;
  mreflexive: ∀a. δ a a ≈ 0;
  msymmetric: ∀a,b. δ a b ≈ δ b a;
  mtineq: ∀a,b,c:ms_carr. δ a b ≤ δ a c + δ c b
}.
```

Note that we call this structure a semi metric space since the reflexivity property is stated in weaker way of the usual δ a b ≈ 0 → a ≈ b (that requires an already existent apartness relation on the elements of the semi metric space).

A semi metric space induces an apartness relation in the following way.

```
definition apart_smetric:= λR.λms:smetric_space R.λa,b:ms.0 < δa b.
lemma apart_of_smetric_space: ∀R.smetric_space R → apartness.
```

To define a semi metric lattice we adopt the same technique we used for divisible and ordered abelian groups. We first define the following technical structure.

```
record mlattice_ (R : todgroup) : Type :={
  ml_mspace_: smetric_space R;
   ml_lattice :> lattice;
  ml_with_: ms_carr ? ml_mspace_ = ap_carr (l_carr  ml_lattice )
}.
```

This time we force the carrier of the semi metric space to be the same same type of the carrier of the apartness embedded in the lattice. We then prove the missing coercion and the final semi metric lattice structure.

```
lemma ml_mspace: ∀ R.mlattice_ R → metric_space R.
record mlattice (R : todgroup) : Type :={
  ml_carr :>mlattice_ R;
  ml_prop1: ∀ a,b:ml_carr. 0 < δ a b → a # b;
  ml_prop2: ∀ a,b,c:ml_carr. δ (a∨ b) (a∨ c) + δ(a∧ b) (a∧ c) ≤ δb c
}.
```

The usual bunch of lemmas to rewrite under δ like x≈z→δ x y≈δ z y are then proved. The following lemma is the key feature of the semi metric lattice structure, relating the order relation induced by the lattice and the metric in a strong way.

```
lemma le_mtri: ∀ R.∀ ml:mlattice R.∀ x,y,z:ml. x ≤ y → y ≤ z → δx z ≈ δx y + δy z.
```

We now have all the algebraic structures needed to prove the sandwich theorem.

3.2.4 The sandwich theorem

To attack the sandwich theorem we first have to define what we mean when we say that a sequence tends to zero. We represent sequences with functions from natural numbers to a space with an order relation, while we say that a sequence tends to zero when given an e positive there exists a natural number N such that for every natural number n greater than N the n-th point of the sequence is between -e and e.

```
definition sequence := λ O:excess.nat → O.
definition tends0 :=
  λ O:pogroup.λ s:sequence O. ∀ e:O.0 < e → ∃ N.∀ n.N < n → −e < s n ∧ s n < e.
```

In Figure 3.2 we report the complete proof of the sandwich theorem. It states that given three sequences an, bn and xn such that for all n, $an_n \leq xn_n \leq bn_n$ then if an and bn tends to an x, then xn also tends to that x.

```
theorem sandwich:
  ∀ R.∀ ml:mlattice R.∀ an,bn,xn:sequence ml.∀ x:ml.
    (∀ n. an n ≤ xn n ∧ xn n ≤ bn n) → an ⇝ x → bn ⇝ x → xn ⇝ x.
```

The proof style is pretty declarative, since we explicitly wrote the term to be obtained in every rewriting step. Thanks to the semantic selection [3] facility of the MATITA user interface it is easy to select meaningful sub-terms of the current goal with the mouse and paste them in the script file.

Lines 1-4 are uninteresting manipulation of the goal and hypothesis. From the fact that an and bn tend to x we extract the witness n1 for an and n2 for bn using e/2 as the chosen epsilon. Note that in our divisible group, division by zero was syntactically forbidden applying the successor function to the divisor, thus e/2 has to be read as $\frac{e}{3}$. Also note that we use the aforementioned divide_preserves_lt lemma to deduce that 0<e/2 from the fact that 0<e.

In line 5 the first proof step is done, we choose the witness $n1 + n2$ that will be used to prove that xn tends to x. We thus obtain an n3 that is bigger than n1+n2 and we are left to prove that -e<δ(xn n3) x and δ(xn n3) x<e.

Lines 6,7 and 8 further manipulate the hypothesis obtaining that δ (an n3) x < e/2, δ (bn n3) x<e/2 and that an n3 ≤ xn n3 ≤ bn n3.

We then proceed by means of forward reasoning, stating the main inequality involved in the lemma:

$$\delta\ xn_{n3}\ x \leq \delta\ bn_{n3}\ x + 2 * \delta\ an_{n3}\ x$$

```
intros (R ml an bn xn x H Ha Hb);
unfold tends0 in Ha Hb ⊢%; unfold d2s in Ha Hb ⊢%; intros (e He);
cases (Ha (e/2) ( divide_preserves_lt  ??? He)) (n1 H1); clear Ha;
cases (Hb (e/2) ( divide_preserves_lt  ??? He)) (n2 H2); clear Hb;
apply (ex_introT ?? (n1+n2)); intros (n3 Lt_n1n2_n3);
lapply (ltwr ??? Lt_n1n2_n3) as Lt_n1n3; lapply (ltwl ??? Lt_n1n2_n3) as Lt_n2n3;
cases (H1 ? Lt_n1n3) (_ daxe); cases (H2 ? Lt_n2n3) (_ dbxe);
cases (H n3) (H7 H8); clear Lt_n1n3 Lt_n2n3 Lt_n1n2_n3 H1 H2 H n1 n2;
cut (δ (xn n3) x ≤ δ (bn n3) x + δ (an n3) x + δ (an n3) x) as main_ineq; [2:
  apply ( le_transitive ???? (mtineq ???? (an n3)));
  cut (δ(an n3) (bn n3)+ −δ(xn n3) (bn n3)≈(δ(an n3) (xn n3))) as H11; [2:
    lapply (le_mtri ????? H7 H8) as H9; clear H7 H8;
    lapply (feq_plusr ? (−δ(xn n3) (bn n3)) ?? H9) as H10; clear H9;
    apply (Eq≈ (0+δ(an n3) (xn n3)) ? (zero_neutral ??));
    apply (Eq≈ (δ(an n3) (xn n3) + 0) ? (plus_comm ???));
    apply (Eq≈ (δ(an n3) (xn n3)+(−δ(xn n3) (bn n3)+δ(xn n3) (bn n3))) ? (opp_inv...
    apply (Eq≈ (δ(an n3) (xn n3)+(δ(xn n3) (bn n3)+ −δ(xn n3) (bn n3))) ? (plus_c...
    apply (Eq≈ ?? (eq_sym ??? (plus_assoc ????))); assumption;]
  apply (Le≪ (δ(an n3) (xn n3)+δ(an n3) x) (msymmetric ??(an n3)(xn n3)));
  apply (Le≪ (δ(an n3) (bn n3)+ −δ(xn n3) (bn n3)+ δ(an n3) x) H11);
  apply (Le≪ (−δ(xn n3) (bn n3)+δ(an n3) (bn n3)+δ(an n3) x) (plus_comm ???));
  apply (Le≪ (−δ(xn n3) (bn n3)+(δ(an n3) (bn n3)+δ(an n3) x)) (plus_assoc ????));
  apply (Le≪ ((δ(an n3) (bn n3)+δ(an n3) x)+ −δ(xn n3) (bn n3)) (plus_comm ???));
  apply lew_opp; [apply mpositive] apply fle_plusr;
  apply (Le≫ ? (plus_comm ???));
  apply (Le≫ (δ(an n3) x+ δx (bn n3)) (msymmetric ????));
  apply mtineq;]
split; [ apply ( lt_le_transitive ????? (mpositive ????));
  apply lt_zero_x_to_lt_opp_x_zero ; assumption;]
apply ( le_lt_transitive ???? main_ineq);
apply (Lt≫ (e/2+e/2+e/2)); [apply (divpow ?e 2)]
repeat (apply ltplus; try assumption);
qed.
```

Figure 3.2: Sandwich theorem proof.

Line 10 uses the triangular inequality property of the order relation we defined on the lattice to change the goal in

$$\delta\ xn_{n3}\ an_{n3} + \delta\ an_{n3}\ x \leq \delta\ bn_{n3}\ x + 2 * \delta\ an_{n3}\ x$$

Next line asserts that

$$\delta\ an_{n3}\ bn_{n3} - \delta\ xn_{n3}\ bn_{n3} \approx \delta\ an_{n3}\ xn_{n3}$$

that can be proved using the le_mtri lemma shown in Section 3.2.3. We then obtain

$$\delta\ an_{n1}\ bn_{n3} \approx \delta\ an_{n3}\ xn_{n3} + \delta\ xn_{n3}\ bn_{n3}$$

using the fact that $an_{n3} \leq xn_{n3} \leq bn_{n3}$. This subproof is closed in line 18, only algebraic manipulation involving the group properties of the semi metric co-domain are used. Almost all rewriting steps are performed deeply, but thanks to coercion trick we are allowed to simply write (opp_inverse ??) instead of

(eq_sym ??? (feq_plusl ???? (opp_inverse ??)))

Lines 19-27 use again trivial properties of the group structure of the semi metric co-domain to perform some rewritings. The msymmetric lemma states that $\delta\ a\ b \approx \delta\ b\ a$ for all a and b. The proof is closed by another application of the triangular inequality. Lines 28 and 29 prove the easy part of the main conjecture, since the metric is always greater then zero and e is positive, -e <δ(xn n3) x.
Last lines prove that δ(xn n3) x < e. The first transitivity step reduces the goal to

$$\delta\ bn_{n3}\ x + 2 * \delta\ an_{n3}\ x < e$$

Then the right hand side is rewritten to $\frac{e}{3} + \frac{e}{3} + \frac{e}{3}$. Every addendum on the left is smaller than every addendum on the right, thus the goal is proved.

Removing the algebraic steps performed rewriting with abelian groups axioms one obtains the usual pen and paper proof, where first an adequate witness is chosen, then thanks to various application of the triangular inequality and the properties of the division operation the goal is solved.

3.3 Methodologies and issues

In this small development we benefit from some features we implemented in Matita during our PHD and that we detail in the second part of the manuscript. The development is reasonably small, consisting of around 200 lemmas and 2000 proof script lines, but is still a sensible test for the coercion subsystem. The number of coercions declared amounts to 70 (counting also composite coercions generated automatically by the system) that his already a high number considering that in the whole fundamental theorem of algebra development [34] the number of coercions amounts to 50.

Coercions have been adopted for two very different purposes: to mimic subtyping between algebraic structures and to overcome the complexity arising from the lack of setoids support in the tactic engine of Matita.

3.3.1 Coercion and inheritance

Figure 3.3 shows the final coercion graph regarding algebraic structures where transitive arrows have been removed.

In the graph there are three diamonds. To make these records behave correctly the system has been modified to exploit the information given by the coercion graph during unification. All the details will be detailed in Chapter 5 but is worth to give the overall intuition here. Consider a formula where both the division operation (defined only on divisible groups) and the less or equal relation, defined only on partially ordered group, appear. The less or equal relation expects an ordered group G as its first argument and elements in og_carr G as its second and third argument. One of these arguments can be a complex expression whose head constant is the division, taking a group Q, a natural number and an element of type dg_carr Q. The output type of the division operation has to be unified with the input type of the less or equal relation. Thus the following unification problem arise:

$$\mathrm{dg\_carr}\ \ ?1 \overset{?}{\equiv} \mathrm{og\_carr}\ \ ?2$$

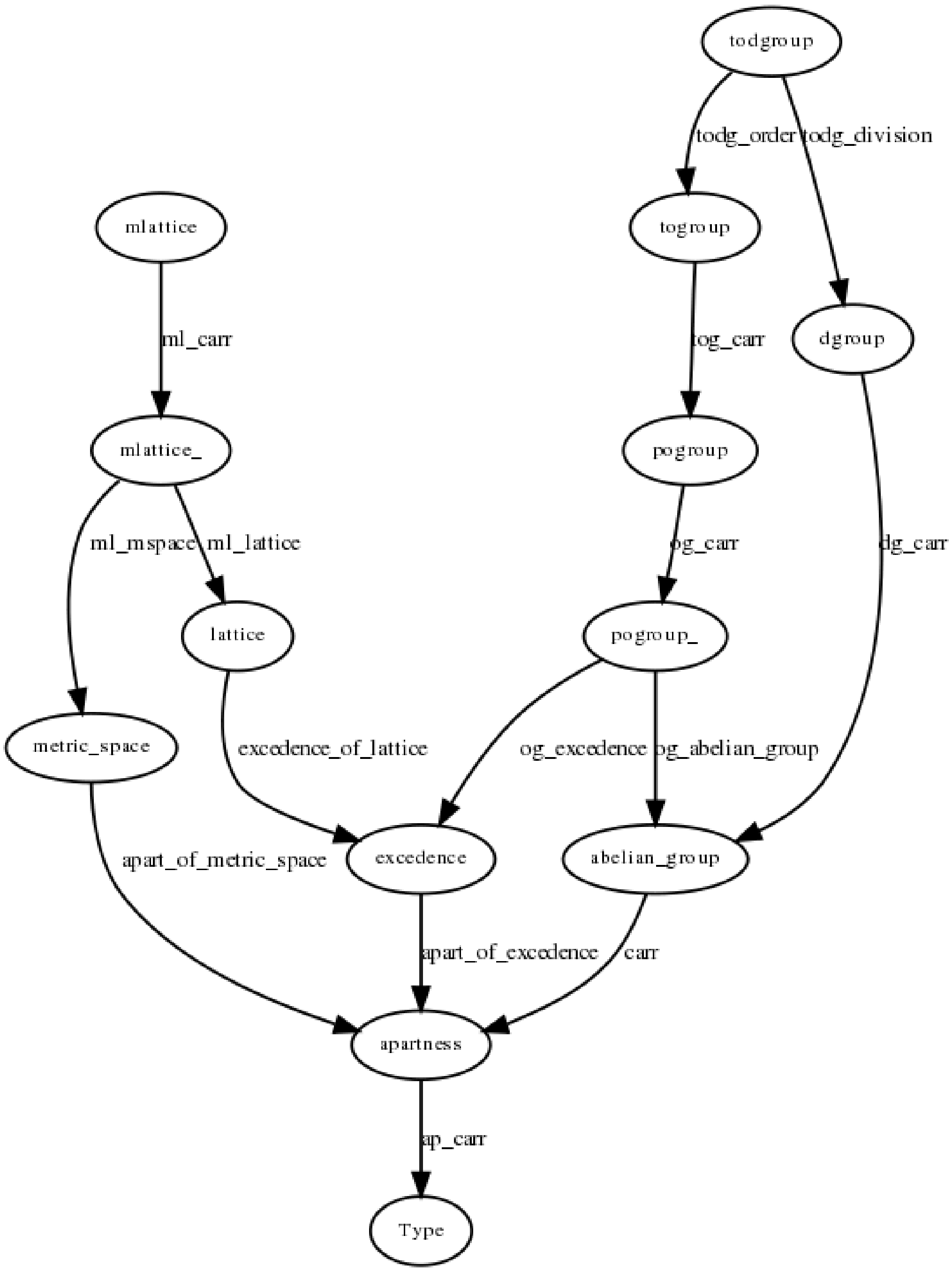


Figure 3.3: Coercion graph in DAMA, transitively reduced

Note that the groups are hidden by the notation, thus they are implicit arguments the system must infer. This unification problem falls in the rigid versus rigid case (δ-expansion is usually avoided as much as possible during conversion/unification since it is too expensive in terms of memory and computational resources). Since the head constants are different a regular unification algorithm would fail. Intuition suggests that the unification problem has to be read as: "Is there an algebraic structure such that it contains a divisible group and an ordered group whose carriers are the same?". The answer is that, if the coercion graph admits a pullback (in categorical sense) for the ordered group and the divisible group then this is the structure we are looking for. Thus the system reduces the former unification problem to the following one:

$$\text{dg\_carr}\ \ (\text{todg\_division}\ \ ?3) \overset{?}{\equiv} \text{og\_carr}\ (\text{tog\_carr}\ (\text{todg\_order}\ ?3))$$

If the todgroup structure has been declared such that the projections todg_division and todg_order return convertible terms (actually types), the unification problem is solved.

The unification algorithm, together with the practice of building structures with multiple inheritance as explained in this chapter, allows to declare complex algebraic hierarchies in an extremely natural way.

3.3.2 Coercion and proof search

During the development, the lack of proper support for setoids in the tactic engine of Matita was leading to incredibly verbose proof scripts where almost one line over two was to elide the context to perform a deep rewriting. Since our PHD was drawing to an end and since the porting of the Setoid Replace Coq tactic was still in progress we decided to abuse the coercion mechanism to perform some steps of proof search.

We do believe that our trick does not scale up as a proper setoid mechanism does. The main cause is that coercions are indexed using only the head constant and thus all lemmas of the form $x \approx y \to C[x] \approx C[y]$ are indistinguishable and the type inference algorithm has to blindly attempt to apply them until one succeeds. Nevertheless, the trick was sufficient to successfully complete this development.

Moreover we think that some of the coercions we declared do really ease life in every formalisation, for example declaring the symmetry property of a predicate as a coercion allows the user to omit from the proof script uninteresting steps. The coercion that most helped is the function law for the group operation $a \approx b \to a + c \approx b + c$ and its variants, that allows to perform deep rewritings under the context $(\cdot + c)$.

3.3.3 Searching and algebraic structures

We already argued that searching facilities are a necessary feature for a user, especially a newcomer. As we will detail in Section 4.3.1 Matita integrates powerful searching facilities. These searching facilities are exploited by many components of the system, one of them being automation. Automatic tactics search the library for relevant lemmas, and use them directly, without requiring the user to list them. The searching algorithm is based on signature similarity. The signature of a goal is the set of constants appearing in the goal itself, and in the context. If the signature contains inductive types it is closed with their constructors, if it contains constructors it is closed with their types. This closure criteria worked pretty nicely until algebraic structures, encoded as records, were used. For example, consider a group, and a goal in which the identity (actually the constant function that projects the identity element) does not appear. Following the criteria just described, no lemmas regarding the identity of the group were found, since the identity was not appearing in the goal signature. Moreover, if the current goal mentions ordered groups, no lemma about abelian group is found since the record type of abelian groups appears nowhere.

We implemented a slightly different algorithm to calculate the signature: when a record projection is found in the goal or in the context, the record type itself and all its projections are also included, and recursively all other record mentioned. This closes the set of lemmas including at least all axioms regarding the algebraic structure under analysis and all the axioms regarding structures from which it inherits. This follows the intuition that every lemma regarding abelian groups can be used on a more specialised structure like ordered groups.

Without this modification, that makes searching facilities aware of the inheritance graph of the algebraic structures, all automatic tactics were pretty useless when coping with algebraic structures.

Part II

A developer perspective

Chapter 4

The proof assistant Matita

We devoted most of the PHD to the design and implementation of the MATITA interactive theorem prover. When our PHD began, MATITA was a bit more then just a nice name for such a tool. It will be described in details in Section 4.1 that many of the fundamental components were almost ready, but there was no usable interface built on top of these functionalities, no real proof was ever formalised with the tool before.

At the very beginning of our PHD we worked closely with S. Zacchiroli [96] to obtain a working interface, a language for writing proofs and tools for managing the library of proved theorems. After this "bootstrapping" phase we mostly dedicated our PHD activity to some aspects of the refinement (type inference) process and automation. Chapter 5 is dedicated to the coercion synthesis mechanism we implemented inside the refiner and the use of that facility in formalising mathematics. Chapter 6 is dedicated to the automatic tactics that we designed and implemented.

The architecture of the system has been described in the published paper "Crafting a Proof Assistant". An overview of the architecture will be given in Section 4.2.

The user interface and the facilities that MATITA provides to the user has been described in the paper " User Interaction with the MATITA Proof Assistant" that we coauthored (see [3]).

Section 4.3.2 details the small step semantic of the MATITA tactic language, to which the author contributed significantly. The paper "Tinycals: Step by Step Tacticals" is the result of such work [81].

Last section (4.4) is dedicated to the problem that afflicted MATITA from its early stage: the difficulty to deliver the system to users. We made a strong contribution to that problem, developing a live-CD that was successfully used during the Types Summer School 2007. Moreover a package suitable for the Debian and Ubuntu Linux distributions has been prepared.

4.1 History

MATITA is an interactive theorem prover being developed by the HELM team at the University of Bologna, under the direction of Prof. Asperti. It may be downloaded at `http://matita.cs.unibo.it`.

The origins of MATITA go back to 1999 when the HELM team was mostly interested in developing document-centric technologies to enhance accessibility on the WWW of distributed libraries of formalised mathematics. At the end of the European project IST-2001-33562 MoWGLI (leaded by Bologna) the following components were ready:

- an XML specification for the Calculus of (Co)Inductive Constructions (CIC), an exportation module for the Coq proof assistant [92], and an independent parser for the exported documents. The exportation of Coq library consists in about 40'000 distinct mathematical objects [80];
- metadata specifications and an innovative tool (WHELP) to index and query the XML knowledge base [1];
- a proof checker Web service (the future kernel of MATITA) to check subsets of the distributed library [95];
- a sophisticated term parser able to deal with potentially ambiguous and incomplete information, typical of the mathematical notation [82];
- a *refiner* component, i.e. a type inference system, based on partially specified terms and unification, used by the disambiguating parser above [80];
- complex transformation algorithms for proof rendering in natural language [33];
- an innovative MathML Presentation rendering widget supporting high-quality bidimensional rendering and visual selection [71].

What was missing to obtain MATITA was to integrate these functionalities, add library management, a proof language, and an authoring interface.

The author began his PHD when the MoWGLI project was almost over. Before that time, his main contribution was the addition of the predicative hierarchy of universes to the proof-checker [91] and some speedup optimisations to the WHELP search engine.

In the following three years the team developed a graphical interface [96, 3] to glue existing components together; support for user defined notation [72] was added; a vernacular of commands for writing and structure proofs was designed and implemented (see Section 4.3.2); automatic tactics (described in Chapter 6) were developed; support for implicit coercions was added (see Chapter 5) and a base library with some interesting result in number theory and typed lambda calculi was developed.

Recently a declarative vernacular [26] has been implemented, and the possibility of generating proof scripts, both declarative and procedural, starting from proof objects is currently under development.

In the last months, a release candidate of the system has been successfully delivered to the students of the Types Summer School through a live CD (see 4.4.2), and the first official version of the system has been released.

4.1.1 Motivations

Our interest and motivation in the development of MATITA relies in the challenging software complexity of this kind of applications. We believe that the interactive theorem proving domain deserves more systems in direct competition to test new ideas and solutions, reaching a critical mass of research teams working in the field. It also looks interesting to have more tools based on similar, partially or totally compatible foundational systems, for exactly the same reason we are interested, say, in different implementations of the same programming language. Finally, several tools as Coq, NuPRL or Mizar have been around for more than 20 years, and their original design has undergone numerous modifications and extensions, often constrained by backward compatibility issues, suggesting that a redesign from scratch can be beneficial.

Mostly due to the circumstances of its origins than to a deliberate choice, MATITA shares the same foundational dialect of Coq, the same implementation language, and it can also directly use results from its library; so MATITA looks like (and partly is) a Coq clone. Then, we deliberately entered in direct competition with Coq adopting the same interaction style of Coq and a similar proof language. However, no code is shared by the two systems and the architectural design as well as the implementative solutions are often different. The result is essentially a lightweight version of Coq, probably very similar to the way Coq itself would look like if entirely rewritten from scratch.

The fact that MATITA and Coq share the same foundational calculus makes them an example (probably the first) of interactive theorem prover that are actually interoperable. MATITA can check Coq proofs and vice versa. This mutually increase the confidence on these systems, following the third-party validation principle: proofs done in a system must be verifiable by a third party tool.

4.2 Architecture

Here we give a brief overview on structure of the software to identify the components that will be discussed in this dissertation. The interested reader can find a precise presentation of the overall architecture of MATITA in [2].

Formulae and proofs are the main data handled by an interactive theorem prover. Both have several possible representations according to the actions performed on them. Each representation is associated with a data type, and the components that constitute an interactive theorem prover can be classified according to the representations they act on. In this section we analyse the architecture of MATITA according to this classification.

The proof and formulae representations used in MATITA as well as its general architecture have been influenced by some design commitments:

1. MATITA is heavily based on the Curry-Howard isomorphism. Execution of procedural and declarative scripts produce proof terms that are kept for later

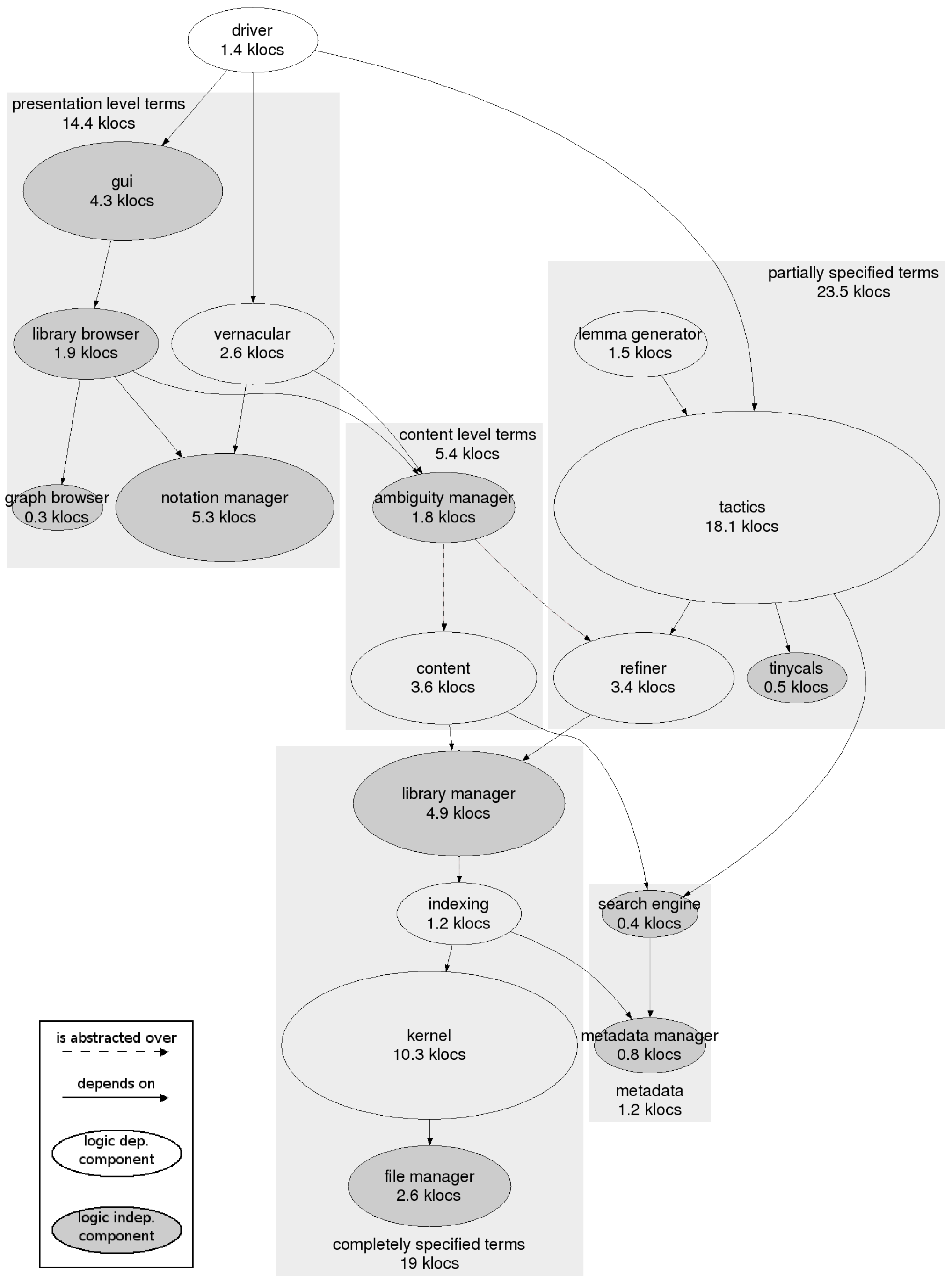


Figure 4.1: MATITA components with thousands of lines of code (*klocs*)

processing. Even incomplete proofs are represented as λ-terms with typed linear placeholders for missing subproofs.

2. The whole library, made of definitions and proof objects only, is searchable and browsable at any time. During browsing proof objects are explained in pseudo-natural language.

3. Proof authoring is performed editing either procedural or declarative scripts. Formulae are typed using ambiguous mathematical notation. Overloading is not syntactically constrained nor avoided using polymorphism.

According to the above commitments, in MATITA we identified 5 term representations: presentation terms (concrete syntax), content terms (abstract syntax trees with overloaded notation), partially specified terms (λ-terms with placeholders), completely specified terms (well typed λ-terms), metadata (approximations of λ-terms).

Figure 4.1 shows the components of MATITA organised according to the term representation they act on. For each component we show the functional dependencies on other components and the number of lines of source code. Dark gray components are either logic independent or can be made such by abstraction. Dashed arrows denote abstractions over logic dependent components. A normal arrow from a logic dependent component to a dark gray one is meant to be a dependency over the component, once it has been instantiated to the logic of the system.

We describe now each term representation together with the components of MATITA acting on them.

4.2.1 Completely Specified Terms

Formalising mathematics is a complex and onerous task and it is extremely important to develop large libraries of "trusted" information to rely on. At this level, the information must be completely specified in a given logical framework in order to allow formal checking. In MATITA proof objects are terms of the Calculus of Inductive Constructions (CIC); terms represent both formulae and proofs. The proof-checker, implemented in the *kernel* component, is a CIC type-checker. Proof objects are saved in an XML format that is shared with the Coq Proof Assistant so that independent verification is possible.

Mathematical concepts (definitions and proof objects) are stored in a distributed library managed by the *file manager*, which acts as an abstraction layer over the concept physical locations.

Concepts stored in the library are indexed for retrieval using metadata. We conceived a logic independent metadata-set that can accommodate most logical frameworks. The logic dependent *indexing* component extracts metadata from mathematical concepts. The logic independent searching tools are described in the next section.

Finally, the *library manager* component is responsible for maintaining the coherence between related concepts (among them automatically generated *lemmas*) and between the different representations of them in the library (as completely specified terms and as metadata that approximate them).

The actual generation of lemmas is a logic dependent activity that is not directly implemented by the library manager, that is kept logic independent: the component provides hooks to register and invoke logic dependent lemma generators, whose implementation is provided in a component that we describe later and that acts on partially specified terms.

4.2.2 Metadata

An extensive library requires an effective and flexible search engine to retrieve concepts. Examples of flexibility are provided by queries up to instantiation or generalisation of given formulae, combination of them with extra-logical constraints such as mathematical classification, and retrieval up to minor differences in the matched formula such as permutation of the hypotheses or logical equivalences. Effectiveness is required to exploit the search engine as a first step in automatic tactics. For instance, a paramodulation based procedure, described in Chapter 6, must first of all retrieve all the equalities in the distributed library that are likely to be exploited in the proof search. Moreover, since search is mostly logic independent, we would like to implement it on a generic representation of formulae that supports all the previous operations.

In MATITA we use relational *metadata* to represent both extra-logical data and a syntactic approximation of a formula (e.g. the constant occurring in head position in the conclusion, the set of constants occurring in the rest of the conclusion and the same information for the hypotheses). The logic dependent *indexing* component, already discussed, generates the syntactic approximation from completely specified terms. The *metadata manager* component stores the metadata in a relational database for scalability and handles, for the library manager, the insertion, removal and indexing of the metadata. The *search engine* component [1] implements the approximated queries on the metadata that can be refined later on, if required, by logic dependent components. More detail on the integrated searching facilities will be discussed in Section 4.3.1

4.2.3 Partially Specified Terms

In partially specified terms, subterms can be omitted replacing them with untyped linear placeholders (called implicit arguments) or with typed metavariables (in the style of [43, 65]). The latter are Curry-Howard isomorphic to omitted subproofs (conjectures still to be proved).

Completely specified terms are often highly redundant to keep the type-checker simple. This redundant information may be omitted during user-machine communication since it is likely to be automatically inferred by the system replacing conversion with unification [89] in the typing rules (that are relaxed to type inference rules). The *refiner* component of MATITA implements unification and the type inference procedure, also inserting implicit coercions [11] to fix local type-checking errors. Coercions are particularly useful in logical systems that lack subtyping [58] and Chapter 5 is dedicated to this argument. The already discussed library manager is also responsible for the management of coercions, that are constants flagged in a special way.

Subproofs are never redundant and if omitted require tactics to instantiate them with partial proofs that have simpler omitted subterms. Tactics are applied to omitted subterms until the proof object becomes completely specified and can be passed to the library manager. Higher order tactics, usually called tacticals and useful to create more complex tactics, are also implemented in the tactics component. The current implementation in MATITA is based on *tinycals* [81], which supports a step-by-step execution of tacticals (usually seen as "black boxes") particularly useful for proof editing, debugging, and maintainability. Tinycals are implemented in MATITA in a small but not trivial component that is completely abstracted on the representation of partial proofs. A brief description of the small-step operational semantic of tynicals is given in Section 4.3.2.

The *lemma generator* component is responsible for the automatic generation of derived concepts (or lemmas), triggered by the insertion of new concepts in the library. The lemmas are generated automatically computing their statements and then proving them by means of tactics or by direct construction of the proof objects.

4.2.4 Content Level Terms

The language used to communicate proofs and especially formulae with the user must also exploit the comfortable and suggestive degree of notational abuse and overloading so typical of the mathematical language. Formalised mathematics cannot hide these ambiguities requiring terms where each symbol has a very precise and definite meaning.

Content level terms provide the (abstract) syntactic structure of the human-oriented (compact, overloaded) encoding. In the *content* component we provide translations from partially specified terms to content level terms and the other way around. The former translation, that loses information, must discriminate between terms used to represent proofs and terms used to represent formulae. Using techniques inspired by [32, 33], the former are translated to a content level representation of proof steps that can in turn easily be rendered in natural language. The representation adopted has greatly influenced the OMDoc [70] proof format that is now isomorphic to it. Terms that represent formulae are translated to MathML Content formulae [62].

The reverse translation for formulae consists in the removal of ambiguity by fixing an interpretation for each ambiguous notation and overloaded symbol used at the content level. The translation is obviously not unique and, if performed locally on each source of ambiguity, leads to a large set of partially specified terms, most of which ill-typed. To solve the problem the *ambiguity manager* component implements an algorithm [82] that drives the translation by alternating translation and refinement steps to prune out ill-typed terms as soon as possible, keeping only the refinable ones. The component is logic independent being completely abstracted over the logical system, the refinement function, and the local translation from content to partially specified terms. The local translation is implemented for occurrences of constants by means of call to the search engine.

The translation from proofs at the content level to partially specified terms is being implemented by means of special tactics following previous work [52, 94] on the implementation of declarative proof styles for procedural proof assistants.

4.2.5 Presentation Level Terms

Presentation level captures the formatting structure (layout, styles, etc.) of proof expressions and other mathematical entities.

An important difference between the content level language and the presentation level language is that only the former is extensible. Indeed, the presentation level language has a finite vocabulary comprising standard layout schemata (fractions, sub/superscripts, matrices, . . .) and the usual mathematical symbols.

The finiteness of the presentation vocabulary allows its standardisation. In particular, for pretty printing of formulae we have adopted MathML Presentation [62], while editing is done using a TEX-like syntax. To visually represent proofs it is enough to embed formulae in plain text enriched with formatting boxes. Since the language of boxes is very simple, many similar specifications exist and we have adopted our own, called BoxML (but we are eager to cooperate for its standardisation with other interested teams).

The *notation manager* component provides the translations from content level terms to presentation level terms and the other way around. It also provides a language [72] to associate notation to content level terms, allowing the user to extend the notation used in MATITA. The notation manager is logic independent since the content level already is.

The remaining components, mostly logic independent, implement in a modular way the user interface of MATITA, that is heavily based on the modern GTK+ toolkit and on standard widgets such as GTKSOURCEVIEW that implements a programming oriented text editor and GTKMATHVIEW that implements rendering of MathML Presentation formulae enabling contextual and controlled interaction with the formula.

The *graph browser* is a GTK+ widget, based on Graphviz, to render dependency graphs with the possibility of contextual interaction with them. It is mainly used in MATITA to explore the dependencies between concepts, but other kind of graphs (e.g. the DAG formed by the declared coercions) are also shown.

The *library browser* is a GTK+ window that mimics a web browser, providing a centralised interface for all the searching and rendering functionalities of MATITA. It is used to hierarchically browse the library, to render proofs and definitions in natural language, to query the search engine, and to inspect dependency graphs embedding the graph browser.

The *GUI* is the graphical user interface of MATITA, inspired by the pioneering work on CtCoq [12] and by Proof General [5]. It differs from Proof General because the sequents are rendered in high quality MathML notation, and because it allows to open multiple library browser windows to interact with the library during proof development.

The hypertextual browsing of the library and proof-by-pointing [14] are both supported by semantic selection. Semantic selection is a technique that consists in enriching the presentation level terms with pointers to the content level terms and to the partially specified terms they correspond to. Highlight of formulae in the widget is constrained to selection of meaningful expressions, i.e. expressions that correspond to a lower level term, that is a content term or a partially or fully specified term. Once the rendering of an upper level term is selected it is possible for the application to retrieve the pointer to the lower level term. An example of applications of semantic selection is *semantic copy & paste*: the user can select an expression and paste it elsewhere preserving its semantics (i.e. the partially specified term), possibly performing some semantic transformation over it (e.g. renaming variables that would be captured or λ-lifting free variables).

Commands to the system can be given either visually (by means of buttons and menus) or textually (the preferred way to input tactics since formulae occurs as tactic arguments). The textual parser for the commands is implemented in the *vernacular* component, that is obviously system (and partially logic) dependent.

To conclude the description of the components of MATITA, the *driver* component, which does not act directly on terms, is responsible for pulling together the other components, for instance to parse a command (using the vernacular component) and then triggering its execution (for instance calling the *tactics* component if the command is a tactic).

4.3 Peculiarities

Even if MATITA can be seen as Coq rewritten from scratch, it includes many distinctive features. While coercions and automation will be described in detail in Section 5 and Section 6 since they are major contributions of the author, here we give a short description of some other peculiarities. A subsection is dedicated to the proof structuring language of MATITA, tinycals, to which the author of this dissertation contributed and that is related to some issues the author mentioned in Chapter 2 Section 2.3.4, but that has been already described in details in [96].

4.3.1 Integrated searching facilities

The first peculiar feature of MATITA, already mentioned in 2.3 is the system-wide integration of the searching facilities of the WHELP [1] search engine.

The metadata model used in WHELP for indexing mathematical notions is essentially based on a single ternary relation $s\mathcal{R}^p\ t$ stating that an object s refers an object t at a given position p. A minimal set of positions is used to discriminate the hypotheses (Hyp), from the conclusion (Concl) and the proof (Proof) of a theorem (respectively, the type of the input parameters, the type of the result, and the body of a definition). Moreover, in the hypothesis and in the conclusion the root position (Main-Hyp and Main-Concl, respectively) is different from deeper positions (that, in a first order setting, essentially amounts to distinguish relational symbols from functional ones).

For example consider the statement:

$$\forall m, n : \mathbb{N}. m \leq n \rightarrow m < (S\ n)$$

its metadata are described by the following table:

Symbol	Position
$\mathbb{N}$	Main-Hyp
$\leq$	Main-Hyp
$<$	Main-Concl
S	Concl

On this metadata model some queries have been implemented. They are mostly independent from the logic (CIC) of MATITA, but some of them actually use some logic dependent functionality of the system (like convertibility) to refine the search result. The most relevant queries are:

match takes as input a type and returns a list of objects (definitions or proofs) inhabiting it. The type of these objects must have the same metadata set as the input type

hint aims to find objects that can be used to prove a given goal in backward fashion. The idea behind this query is to use the metadata model to find, given a goal g, all objects whose type $t_1 \to t_2 \to \cdots \to t_n \to t$ such that there exists a substitution σ that makes the following equation valid: $t\theta = g$. A necessary condition, if we do not consider reduction, is that the set of constants in t must be a subset of those in g. In terms of our metadata model, the problem consists to find all s such as $\{x|Ref(C, s, x)\} \subseteq A$ where A is the set of constants in g. The interested reader can find a more detailed description of this query in [1].

locate implements a simple "lookup by name" for library notions. Once fed with an identifier i, the query returns the list of all objects whose name is i. The query support wild cards, allowing to query for objects whose name contains a given string or ends with a given suffix. This query does not exploit the metadata model previously described, but just the fact that all objects are listed in one single place (the database actually).

The **match** query is triggered when, in interactive mode, the user defines a new objects (or claims a new result). The resulting objects types are compared (actually converted) with the type specified by the user. This implements a duplicate check, that warns the user that his claim is already proved in the library. The user can declare her intention to prove again an already existing object (actually generating what is called a variant) and in this case the system does not complain.

hint is both available as an interactive command, that the user can run to know the list of lemmas that can be applied to the current goal, and a batch procedure that has been used to implement Prolog style automatic tactics.

The **locate** tactic, in conjunction with a consistent naming policy, thanks to the wild cards facility, can really be effective. Moreover it can be automatically used to inform the user that lemmas with the same (or similar) name are already part of the library.

All these searching facilities are integrated in the system, that stores (and removes) metadata in a relational database when the user defines (or deletes) an object.

Using the same metadata model some other facilities have been implemented. For example the user may ask to know all objects that are using the notion he is currently pondering if it is worth modifying.

4.3.2 Tinycals

tinycals [81] can be seen an alternative to a subset of LCF tacticals, that being executed in small step fashion, allow the user to better read and structure her proof script.

We already commented on the bad impact the commonly implemented big step execution of tacticals interacts with the good practice of structuring scripts an re-reading them in Section 2.3.4.

What follows is a rework with minor improvements of the formal semantic described in [81].

Syntax and semantics

The grammar of tinycals is reported in Table 4.1, where $\langle L \rangle$ is the top-level nonterminal generating the script language. $\langle L \rangle$ is a sequence of statements $\langle S \rangle$. Each statement is either an atomic tactical $\langle B \rangle$ (marked with "`tactic`") or a tinycal.

Note that the part of the grammar related to the tinycals themselves is completely de-structured. The need for embedding the structured syntax of LCF tacticals (nonterminal $\langle B \rangle$) in the syntax of tinycals is due to the fact that not all LCF tacticals can be executed in a small step fashion, thus for some of them, and the ones that are combined by means these big step tacticals, the syntax has to be structured. See the end of Section 4.3.2 for the explanation for this limitation.

For the time being, the reader can suppose the syntax to be restricted to the case $\langle B \rangle ::= \langle T \rangle$.

We will now describe the semantics of tinycals which is parametric in the proof status tactics act on and also in their semantics (see Table 4.2).

Table 4.1: Abstract syntax of tinycals and core LCF tacticals.

$\langle S\rangle$	::=		(**statements**)	$\langle L\rangle$	::=		(**language**)
		“`tactic`” $\langle B\rangle$	(tactic)			$\langle S\rangle$	(statement)
$\mid$		“**.**”	(dot)	$\mid$		$\langle S\rangle$ $\langle S\rangle$	(sequence)
$\mid$		“**;**”	(semicolon)	$\langle B\rangle$	::=		(**tacticals**)
$\mid$		“**[**”	(branch)			$\langle T\rangle$	(tactic)
$\mid$		“**|**”	(shift)	$\mid$		“`try`” $\langle B\rangle$	(recovery)
$\mid$		$i_1,\ldots,i_n$“**:**”	(projection)	$\mid$		“`repeat`” $\langle B\rangle$	(looping)
$\mid$		“ $*$**:**”	(wild card)	$\mid$		$\langle B\rangle$“**;**”$\langle B\rangle$	(composition)
$\mid$		“`skip`”	(acknowledge)	$\mid$		$\langle B\rangle$“**;[**”	(branching)
$\mid$		“**]**”	(merge)			$\langle B\rangle$“**|**” … “**|**”$\langle B\rangle$“**]**”	
$\mid$		“`focus`” $[g_1;\cdots;g_n]$	(selection)	$\langle T\rangle$	::= …		(**tactics**)
$\mid$		“`done`”	(de-selection)				

A *proof status* is the logical status of the current proof. It can be seen as the current proof tree, but there is no need for it to actually be a tree. MATITA for instance just keeps the set of conjectures to prove, together with a proof term where meta-variables occur in place of missing subparts. From a semantic point of view the proof status is an abstract data type. Intuitively, it must describe at least the set of conjectures yet to be proved. A *Goal* is another abstract data type used to index conjectures.

Table 4.2: Semantics parameters.

proof status:	ξ
proof goal:	*goal*
tactic application:	$\mathit{apply\_tac} : T \to \xi \to \mathit{goal} \to \xi \times \mathit{goal}$ `list` $\times$ goal `list`

The function *apply_tac* implements tactic application. It consumes as input a tactic, a proof status, and a goal (the conjecture the tactic should act on), and returns as output a proof status and two lists of goals: the set of newly opened goals and the set of goals which have been closed. This choice enables our semantics to account for *side-effects*, that is: tactics can close goals other than that on which they have been applied, a feature implemented in several proof assistants via existential or meta-variables [43, 65]. The proof status was not directly manipulated by tactics in LCF because of the lack of meta-variables and side effects.

In the rest of this section we will define the semantics of tinycals as a transition (denoted by $\longrightarrow$) on evaluation status. *Evaluation status* are defined in Table 4.3.

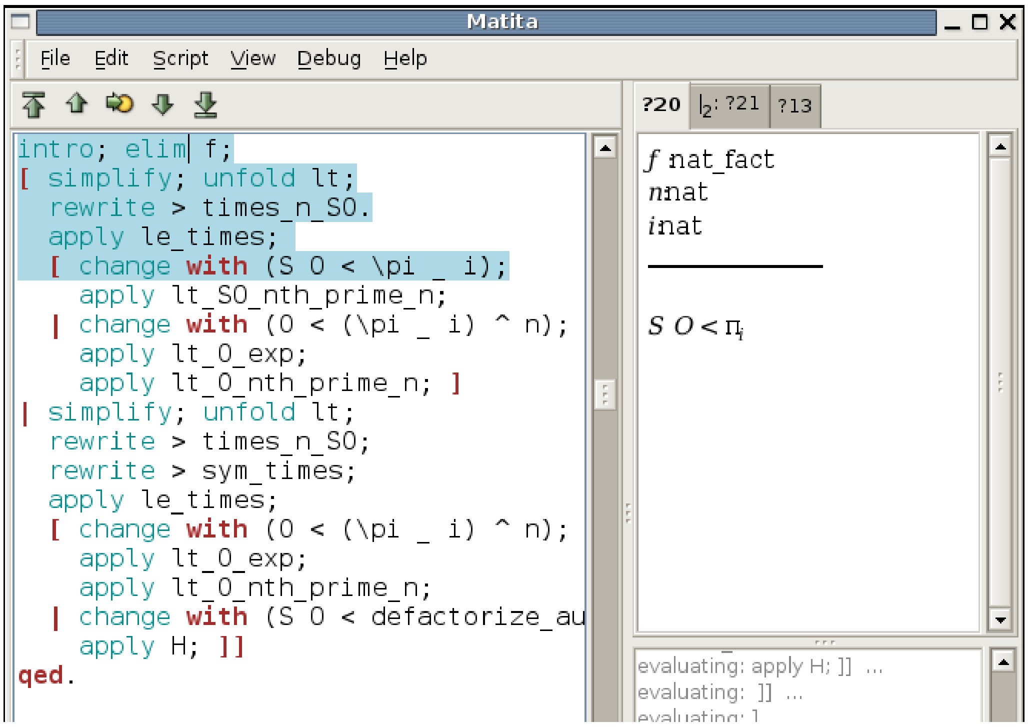
Table 4.3: Evaluation status.

task	$=$	$\mathtt{int} \times (\mathtt{Open}\ \mathit{goal} \mid \mathtt{Closed}\ \mathit{goal})$	(task)
Γ	$=$	$\mathit{task}\ \mathtt{list}$	(context)
τ	$=$	$\mathit{task}\ \mathtt{list}$	(“todo” list)
κ	$=$	$\mathit{task}\ \mathtt{list}$	(dot’s continuations)
$\mathit{ctxt\_tag}$	$=$	$\mathtt{B} \mid \mathtt{F}$	(stack level tag)
$\mathit{ctxt\_stack}$	$=$	$(\Gamma \times \tau \times \kappa \times \mathit{ctxt\_tag})\ \mathtt{list}$	(context stack)
code	$=$	$\langle S \rangle\ \mathtt{list}$	(statements)
status	$=$	$\mathit{code} \times \xi \times \mathit{ctxt\_stack}$	(evaluation status)

The first component of the status (*code*) is a list of statements of the tinycals grammar. The list is consumed, one statement at a time, by each transition. This choice has been guided by the un-structured form of our grammar and is the heart of the fine-grained execution of tinycals.

The second component is the proof status, which we enrich with a *context stack* (the third component). The context stack, a representation of the proof history so far, is handled as a stack: levels get pushed on top of it either when the branching tinycal “[” is evaluated, or when “`focus`” is; levels get popped out of it when the corresponding closing tinycals are (“]” for “[” and “`done`” for “`focus`”). Since the syntax is un-structured, we can not ensure statically proper nesting of tinycals, therefore each stack level is equipped with a *tag* which annotates it with the creating tinycal (`B` for "[" and `F` for "`focus`"). In addition to the tag, each stack level has three components Γ, τ and κ respectively for active tasks, tasks postponed to the end of branching and tasks postponed by "**.**". The role of these components will be explained in the description of the tinycals that acts on them. Each component is a sequence of numbered tasks. A *task* is an handler to either a conjecture yet to be proved, or one which has been closed by a side-effect. In the latter case the user will have to confirm the instantiation with "`skip`".

Each evaluation status is meaningful to the user and can be presented by slightly modifying preexisting user interfaces.



```
intro; elim f;
[ simplify; unfold lt;
  rewrite > times_n_SO.
  apply le_times;
  [ change with (S O < \pi _ i);
    apply lt_SO_nth_prime_n;
  | change with (O < (\pi _ i) ^ n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n; ]
| simplify; unfold lt;
  rewrite > times_n_SO;
  rewrite > sym_times;
  apply le_times;
  [ change with (O < (\pi _ i) ^ n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n;
  | change with (S O < defactorize_au
    apply H; ]]
qed.
```

f : nat_fact
n : nat
i : nat

$S\ O < \pi_i$

```
evaluating: apply H; ]]  ...
evaluating:  ]]  ...
evaluating: ]
```

Figure 4.2: MATITA user interface in the middle of a proof

Our presentation choice is described is can be seen in Figure 4.2, where the interesting part of the proof status is presented as a notebook of conjectures to prove, and the conjecture labels represent the relevant information from the context stack by means of: 1) bold text (for conjectures in the currently selected branches, targets of the next tactic application; they are kept in the Γ component of the top of the stack); 2) subscripts (for not yet selected conjectures in sibling branches; they are kept in the Γ component of the level below the top of the stack).

The rest of the information hold in the stack does not need to be shown to the user since it does not affect immediate user actions.

We describe first the semantics of the tinycals that do not involve the creation of new levels on the stack. The semantics is shown in Tables 4.4 and 4.5.

Table 4.4: Basic tinycals semantics (1 of 2).

$$\langle \text{“}\texttt{tactic}\text{”}\ \langle T \rangle :: c, \xi, \langle \Gamma, \tau, \kappa, t \rangle :: S \rangle \ \longrightarrow\ \langle c, \xi_n, S' \rangle \qquad n \geq 1$$

where $[g_1; \cdots; g_n] = \mathit{get\_open\_goals\_in\_tasks\_list}(\Gamma)$

$$\text{and } \begin{cases} \langle \xi_0, G^o_0, G^c_0 \rangle = \langle \xi, [\,], [\,] \rangle \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1} \rangle = \langle \xi_i, G^o_i, G^c_i \rangle & g_{i+1} \in G^c_i \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1} \rangle = \langle \xi', (G^o_i \setminus G^c) \cup G^o, G^c_i \cup G^c \rangle & g_{i+1} \notin G^c_i \\ \quad \text{where } \langle \xi', G^o, G^c \rangle = \mathit{apply\_tac}(T, \xi_i, g_{i+1}) \end{cases}$$

and $S' = \langle \Gamma', \tau', \kappa', t \rangle :: \mathit{close\_tasks}(G^c_n, S)$

and $\Gamma' = \mathit{mark\_as\_handled}(G^o_n)$

and $\tau' = \mathit{remove\_tasks}(G^c_n, \tau)$

and $\kappa' = \mathit{remove\_tasks}(G^c_n, \kappa)$

$$\langle \text{“}\texttt{;}\text{”} :: c, \xi, S \rangle \ \longrightarrow\ \langle c, \xi, S \rangle$$

The utility functions used in the description of the semantic are reported at the end of Section 8.2.

Tactic application Consider the first case of the tinycals semantics of Table 4.4. It makes use of the first component (denoted Γ) of a stack level, which represent the “current” goals, that is the set of goals to which the next tactic evaluated will be applied.

Table 4.5: Basic tinycals semantics (2 of 2).

$$\langle \text{“}\texttt{skip}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \quad \longrightarrow \quad \langle c, \xi, S'\rangle$$

where $\Gamma = [\langle j_1, \texttt{Closed}\ g_1\rangle; \cdots; \langle j_n, \texttt{Closed}\ g_n\rangle]$ $\qquad n \geq 1$

and $G^c = [g_1; \cdots; g_n]$

and $S' = \langle [\,], \mathit{remove\_tasks}(G^c, \tau), \mathit{remove\_tasks}(G^c, \kappa), t\rangle$
$:: \mathit{close\_tasks}(G^c, S)$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \quad \longrightarrow \quad \langle c, \xi, \langle [l_1], \tau, [l_2; \cdots; l_n] \cup \kappa, t\rangle :: S\rangle \quad n \geq 1$$

where $\mathit{get\_open\_tasks}(\Gamma) = [l_1; \cdots; l_n]$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, l :: \kappa, t\rangle :: S\rangle \quad \longrightarrow \quad \langle c, \xi, \langle [l], \tau, \kappa, t\rangle :: S\rangle$$

where $\mathit{get\_open\_tasks}(\Gamma) = [\,]$

When a tactic is evaluated, the set Γ of current goals is inspected (expecting to find at least one of them), and the tactic is applied in turn to each of them in order to obtain the final proof status. At each step i the two sets C_i^o and G_i^c of goals opened and closed so far are updated. This process is atomic to the user (i.e. no feedback is given while the tactic is being applied to each of the current goals in turn), but she is free to cast off atomicity using branching. After the tactic has been applied to all goals, the new set of current goals is created containing all the goals which have been opened during the applications, but not already closed. They are marked (using the *mark_as_handled* utility) so that they do not satisfy the *unhandled* predicate, indicating that some tactic has been applied to them. Goals closed by side effects are removed from τ and κ and marked as `Closed` in S. The reader can find a detailed description of this procedure at the end of Section 8.2.

Sequential composition Since sequencing is handled by Γ, the semantics of “**;**” is simply the identity function. We kept it in the syntax of tinycal for preserving the parallelism with LCF tacticals.

Side-effects handling "`skip`" (first case in Table 4.5) is a tinycal used to deal with side-effects. Consider for instance the case in which there are two current goals on which the user branches. It can happen that applying a tactic to the first one closes the second, removing the need of the second branch in the script. Using tinycals the user will never see branches she was aware of disappear without notice. Cases like the above one are thus handled marking the branch as `Closed` (using the *close_tasks* utility) on the stack and requiring the user to manually acknowledge what happened on it using the "`skip`" tinycal, preserving the correspondence between script structure and proof tree.

Consider the following script:

```
apply trans_eq; [ apply H | apply H1 | skip ]
```

where the application of the transitivity property of equality to the conjecture $L = R$ opens the three conjectures $?_1$: L=$?_3$, $?_2$: $?_3$=R and $?_3$: nat. Applying the hypothesis H instantiates $?_3$, implicitly closing the third conjecture, that thus has to be acknowledged.

Local de-structuring Structuring proof scripts enhances their readability as long as the script structure mimics the structure of the intuition behind the proof. For this reason, authors do not always desire to structure proof scripts down to the most far leaf of the proof tree.

Consider for instance the following script snippet template:

```
tac1;
 [ tac2. tac3.
 | tac4; [ tac5 | tac6 ] ]
```

Here the author is trying to mock-up the structure of the proof (two main branches, with two more branches in the second one), without caring about the structure of the first branch.

Lcf tacticals do not allow un-structured scripts to be nested inside branches. In the example, they would only allow to replace the first branch with the identity tactic, continuing the un-structured snippet "tac2. tac3." at the end of the outermost branching tactical, but this way the correspondence among script structure and proof tree would be completely lost. The semantics of the tinycal "**.**" (last two cases of Table 4.5) accounts for local use of un-structured script snippets.

When "**.**" is applied to a non-empty set of current goals, the first one is selected and become the new singleton current goals set Γ. The remaining goals are remembered in the third component of the current stack level (*dot's continuations*, denoted κ), so that when the "**.**" is applied again on an empty set of goals they can be recalled in turn. The locality of "**.**" is inherited by the locality of dot's continuation κ to stack levels.

Table 4.6: Branching tinycals semantics (1 of 2).

$$\langle \text{“[”} :: c, \xi, \langle [l_1;\cdots;l_n], \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 2$$

$$\text{where } \mathit{renumber\_branches}([l_1;\cdots;l_n]) = [l'_1;\cdots;l'_n]$$

$$\text{and } S' = \langle [l'_1], [\,], [\,], \mathtt{B}\rangle :: \langle [l'_2;\cdots;l'_n], \tau, \kappa, t\rangle :: S$$

$$\langle \text{“|”} :: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 1$$

$$\text{where } S' = \langle [l_1], \tau \cup \mathit{get\_open\_tasks}(\Gamma) \cup \kappa, [\,], \mathtt{B}\rangle :: \langle [l_2;\cdots;l_n], \tau', \kappa', t'\rangle :: S$$

$$\langle i_1,\ldots,i_n \text{“:”} :: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

$$\text{where } \mathit{unhandled}(l)$$

$$\text{and } \forall j = 1 \ldots n, \quad \exists l_j = \langle j, s_j\rangle, \quad l_j \in l :: \Gamma'$$

$$\text{and } S' = \langle [l_1;\cdots;l_n], \tau, [\,], \mathtt{B}\rangle :: \langle (l :: \Gamma') \setminus [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S$$

Tables 4.6 and 4.7 describe the semantics of tinycals that require a stack discipline.

Branching Support for branching is implemented by "[", which creates a new level on the stack for the first of the current goals. Remaining goals (the current *branching context*) are stored in the level just below the freshly created one. There are three

Table 4.7: Branching tinycals semantics (2 of 2).

$\langle \text{“}*\texttt{:}\text{”} :: c, \xi, \langle [l], \tau, [\,], \texttt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \quad\longrightarrow\quad \langle c, \xi, S'\rangle$

where $unhandled(l)$

and $S' = \langle l :: \Gamma', \tau, [\,], \texttt{B}\rangle :: \langle [\,], \tau', \kappa', t'\rangle :: S$

$\langle \text{“]”} :: c, \xi, \langle \Gamma, \tau, \kappa, \texttt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \quad\longrightarrow\quad \langle c, \xi, S'\rangle$

where $S' = \langle \tau \cup get\_open\_tasks(\Gamma) \cup \Gamma' \cup \kappa, \tau', \kappa', t'\rangle :: S$

$\langle \text{“}\texttt{focus}\text{”}\ [g_1;\cdots; g_n] :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \quad\longrightarrow\quad \langle c, \xi, S'\rangle$

where $g_i \in get\_open\_goals\_in\_status(S)$

and $S' = \langle mark\_as\_handled([g_1; \cdots; g_n]), [\,], [\,], \texttt{F}\rangle$

$:: close\_tasks(\langle \Gamma, \tau, \kappa, t\rangle :: S)$

$\langle \text{“}\texttt{done}\text{”} :: c, \xi, \langle [\,], [\,], [\,], \texttt{F}\rangle :: S\rangle \quad\longrightarrow\quad \langle c, \xi, S\rangle$

different ways of selecting them. Repeated uses of “|” consume the branching context in sequential order. $i_1,\ldots,i_n$“**:**” enables multiple positional selection of goals from the branching context. “$*$**:**” recall all goals of the current branching context as the new set of current goals. The semantics of all these branching tacticals is shown in Table 4.6 the first case of Table 4.7.

Each time the user finishes working on the current goals and selects a new goal from the branching context, the result of her work (namely the current goals in Γ) needs to be saved for restoring at the end of the branching construct. This is needed to implement the LCF semantics that provides support for snippets like the following:

```
tac1; [ tac2 | tac3 ]; tac4
```

where the goals resulting by the application of `tac2` *and* `tac3` are re-flowed together to create the goals set for `tac4`.

The place where we store them is the second component of stack levels (*todo list*, denoted τ). Each time a branching selection tinycal is used the current goals set (possibly empty) is appended to the todo list for the current stack level.

When "]" is used to finish branching (second rule of Table 4.7), the todo list τ is used to create the new set of current goals Γ, together with the goals not handled during the branching (note that this is a small improvement over LCF tactical semantics, where leaving not handled branches is not allowed).

Since "[" already provides for the creation of the first branch, in Table 4.6 there is some additional machinery to distinguish fresh goals (goals on which the user has not acted yet) from non-fresh one. This distinction is used to decide whether to drop the current branch form the branching context when "|" or $i_1,\ldots,i_n$"`:`" are used.

Focusing The pair of tinycals "`focus`"... "`done`" is similar in spirit to the pair "["... "]", but is not required to work on the current branching context. With "`focus`", goals located everywhere on the stack can be recalled to form a new set of current goals. On this the user is then free to work as she prefer, for instance branching, but is required to close all of them before invoking "`done`".

The intended use of "`focus`"... "`done`" is to deal with meta-variables and side effects. The application of a tactic to a conjecture with meta-variables in the conclusion or hypotheses can instantiate the meta-variables making other conjectures false. In other words, in presence of meta-variables conjectures are no longer independent and it becomes crucial to consider and close a bunch or dependent conjectures together, even if in far away branches of the proof. In these cases "`focus`"... "`done`" is used to select all the related branches for immediate work on them. Alternatively, "`focus`"... "`done`" can be used to jump on a remote branch of the tree in order to instantiate a meta-variable by side effects before resuming proof search from the current position.

Note that using "`focus`"... "`done`", no harm is done to the proper structuring of scripts, since all goals the user is aware of, if closed, will be marked as `Closed` requiring her to manually "`skip`" them later on in the proof.

Digression on other tacticals

Of the basic LCF tacticals, we have considered so far only sequential composition and branching. It is worth discussing the remaining ones, in particular *try*, $||$ (or-else) and *repeat*.

The *try* T tactical, that never fails, applies the tactic T, behaving as the identity if T fails. It is a particular case of the or-else tactical: $T_1||T_2$ behaves as T_1 if T_1 does not fail, as T_2 otherwise. Thus *try* T is equivalent to $T||id$.

The try and or-else tacticals occur in a script with two different usages. The most common one is after sequential composition: T_1; *try* T_2 or $T_1; T_2||T_3$. Here the idea is that the user knows that T_2 can be applied to some of the goals generated by T_1 (and T_3 to the others in the second case). So she is faced with two possibilities: either use branching and repeat T_2 (or T_3) in every branch, or use sequential composition and backtracking (encapsulated in the two tacticals). Tinycals offer a better solution to either choice by means of the projection and wild card tinycals: $T_1; [i_1, \ldots, i_n : T_2|* : T_3]$. The latter expression is not also more informative to the reader, but it is also computationally more efficient since it avoids the (maybe costly) application of T_2 to several goals.

The second usage of try and or-else is inside a repeat tactical. The *repeat* T tactical applies T once, failing if T fails; otherwise the tactical recursively applies T again on every goal opened by T until T fails, in which case it behaves as the identity tactic.

Is it possible to provide an un-structured version of *try* T, $T||T'$, and *repeat* T in the spirit of tinycals in order to allow the user to write and execute T step by step inspecting the intermediate evaluation status? The answer is negative as we can easily see in the simplest case, that of *try* T. Consider the statement T; *try* $(T_1; T_2)$ where sequential composition is supposed to be provided by the corresponding tinycal. Let T open two goals and suppose that "*try*" is executed atomically so that the evaluation point is just before T_1. When the user executes T_1, T_1 can be applied as expected to both goals in sequence. Let ξ be the proof status after the application of T and let ξ_1 and ξ_2 be those after the application of T_1 to the first and second goal respectively. Let now the user execute the identity tinycal ";" followed by T_2 and let T_2 fail over the first goal. To respect the intended semantics of the tactical, the status ξ_2 should be *partially* backtracked to undo the changes from ξ to ξ_1, preserving those from ξ_1 to ξ_2.

If the system has side effects the latter operation is undefined, since T_1 applied to ξ could have instantiated meta-variables that controlled the behaviour of T_1 applied to ξ_1. Thus undoing the application of T_1 to the first goal also invalidates the previous application of T_1 to the second goal.

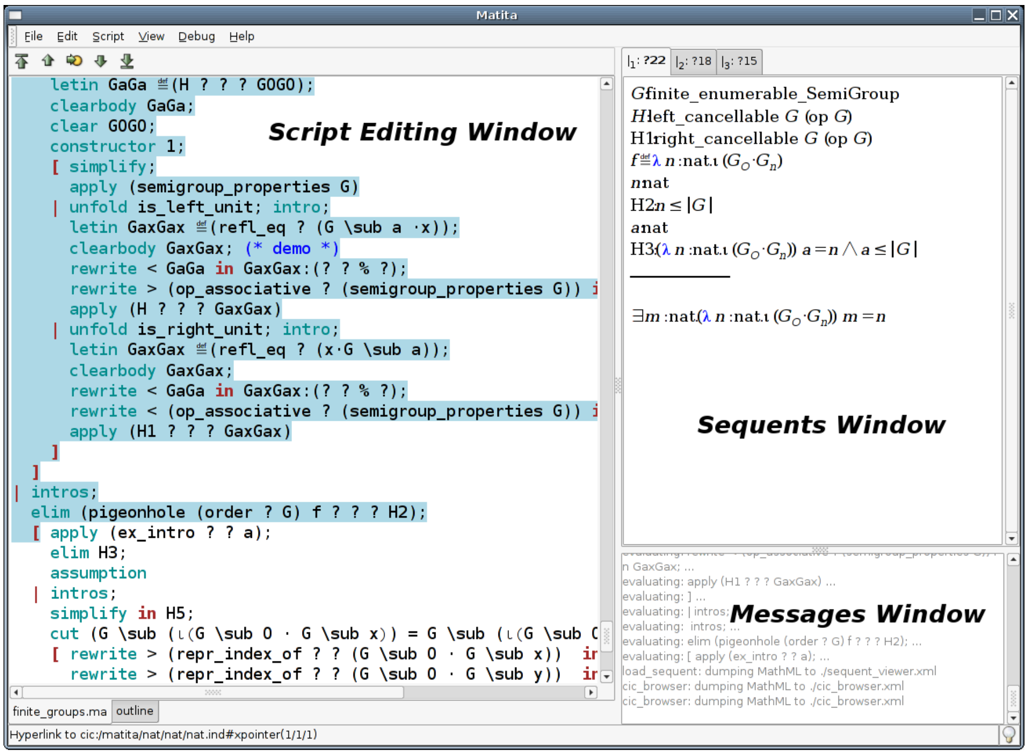
Even if the system has no side effects, the requirement that proof status can be partially backtracked is quite restrictive on the possible implementations of a proof status. For instance, a proof status cannot be a simple proof term with occurrences of meta-variables in place of conjectures, since backtracking a tactic would require the replacement of a precise subterm with a meta-variable, but there would be no information to detect which subterm.

As a final remark, the simplest solution of implementing partial backtracking by means of a full backtrack to ξ followed by an application of T_1 to the second goal only does not conform to the spirit of tinycals. With this implementation, the application of T_1 to the second goal would be performed twice, sweeping the waste of computational resources under the rug. The only honest solution consists of keeping all tacticals, except branching and sequential composition, fully structured as they are now. The user that wants to inspect the behaviour of $T; \mathit{try}\ T_1$ before that of $T; \mathit{try}\ (T_1; T_2)$ is obliged to do so by executing atomically $\mathit{try}\ T_1$, backtracking by hand and executing $\mathit{try}\ (T_1; T_2)$ from scratch. A similar conclusion is reached for the remaining tacticals. For this reason in the syntax given in Table 4.1 the production $\langle B \rangle$ lists all the traditional tacticals that are not subsumed by tinycals. Notice that atomic sequential composition and atomic branching (as implemented in the previous section) are also listed since tinycals cannot occur as arguments of a tactical.

4.3.3 Other peculiarities

All the peculiarities described so far had a contribution from the author of this dissertation. Here we briefly describe other features that characterise the system.

The MoWGLI project developed rendering techniques for mathematical documents. These have been reused in MATITA. Figure 4.3 shows the Math-ML rendering widget (supporting bi-dimensional notations) used in the sequent window. This



```
    letin GaGa ≝(H ? ? ? GOGO);
    clearbody GaGa;
    clear GOGO;
    constructor 1;
    [ simplify;
      apply (semigroup_properties G)
    | unfold is_left_unit; intro;
      letin GaxGax ≝(refl_eq ? (G \sub a ·x));
      clearbody GaxGax; (* demo *)
      rewrite < GaGa in GaxGax:(? ? % ?);
      rewrite > (op_associative ? (semigroup_properties G)) i
      apply (H ? ? ? GaxGax)
    | unfold is_right_unit; intro;
      letin GaxGax ≝(refl_eq ? (x·G \sub a));
      clearbody GaxGax;
      rewrite < GaGa in GaxGax:(? ? % ?);
      rewrite < (op_associative ? (semigroup_properties G)) i
      apply (H1 ? ? ? GaxGax)
    ]
  ]
| intros;
  elim (pigeonhole (order ? G) f ? ? ? H2);
  [ apply (ex_intro ? ? a);
    elim H3;
    assumption
  | intros;
    simplify in H5;
    cut (G \sub (ι(G \sub 0 · G \sub x)) = G \sub (ι(G \sub C
    [ rewrite > (repr_index_of ? ? (G \sub 0 · G \sub x))  ir
      rewrite > (repr_index_of ? ? (G \sub 0 · G \sub y))  ir
```

$|_1$: ?22 $|_2$: ?18 $|_3$: ?15

Gfinite_enumerable_SemiGroup
Hleft_cancellable G (op G)
H1right_cancellable G (op G)
$f \stackrel{def}{=} \lambda\, n$:nat.ι $(G_O \cdot G_n)$
$n$nat
H2$n \le |G|$
$a$nat
H3($\lambda\, n$:nat.ι $(G_O \cdot G_n)$) $a = n \wedge a \le |G|$

$\exists m$:nat.($\lambda\, n$:nat.ι $(G_O \cdot G_n)$) $m = n$

```
n GaxGax; ...
evaluating: apply (H1 ? ? ? GaxGax) ...
evaluating: ] ...
evaluating: | intros; ...
evaluating:  intros; ...
evaluating: elim (pigeonhole (order ? G) f ? ? ? H2); ...
evaluating: [ apply (ex_intro ? ? a); ...
load_sequent: dumping MathML to ./sequent_viewer.xml
cic_browser: dumping MathML to ./cic_browser.xml
cic_browser: dumping MathML to ./cic_browser.xml
```

Hyperlink to cic:/matita/nat/nat/nat.ind#xpointer(1/1/1)

Figure 4.3: MATITA main window

widget also support semantic selection (only structurally meaningful sub-terms can be be selected) and hyperlinks, allowing the user to reach the definition of an object with just one click.

Another feature inherited from the MoWGLI project is the natural language rendering of proof objects. While the user builds a proof with tactics a window can display the rendering of the proof object constructed so far in natural language. This (pseudo) natural language gave the basis for a declarative language, currently under development [26], that can be executed by the system as well as generated using the aforementioned natural language rendering facility. We will use this facility in Chapter 6 to obtain a nice proof script from the proof object generated by an automatic tactic that performs rewriting. The possibility of generating a procedural proof script starting from a proof object is also under development [49].

4.4 Delivering the system

MATITA always suffered the problem of being hard to deliver to the users (like students or researchers). The main cause of that problem is the fact that MATITA uses a relational database management system (DBMS) to store metadata. The second difficulty is that MATITA has been written reusing many components already available, thus depends on many external libraries.

During the MoWGLI project a huge set of metadata, corresponding to the whole COQ library (contributions included), was used. The amount of indexed objects was around 40,000 and the biggest relation (briefly explained in 4.3.1) counted more than 850,000 entries. Due to the size of such dataset a scalable relational DBMS was adopted, namely MySQL. When this technology was integrated in MATITA it implied a dependency over MySQL, that is a complex tool that needs a proper configuration to work. This has always hindered the delivery of the system.

Although installing external libraries (like the Math-ML rendering widget) is just a matter of time, the high number of such dependencies has always been a deterrent for users interested to give the system a try. Moreover, the system has always been developed on computers running the Debian GNU/Linux operating system. The author of this thesis and another team component (Stefano Zacchiroli) are Debian developers, and contributed to the packaging of all needed libraries. This allows an easy and smooth installation of all that libraries on a computer running Debian. Not surprisingly, this has proven not to be a complete solution to that problem, since Debian has never been a widespread distribution for personal Desktop/Laptop computers.

When our advisor Andrea Asperti decided to organise the 2007 Types Summer School in Bologna[1], obviously giving some lectures on MATITA, these issues had to be solved once for all. In addition to that, many other systems developed by the Types community suffer from similar problems, requiring uncommon libraries of specific versions of some external softwares to run properly.

To solve the first issue regarding MATITA we reworked the database related subsystem (see Section 4.4.1). To solve the second issue, that regards not only MATITA but is generically related to all systems, we developed ITPLive!. ITPLive![2] is a CD that contains a bootable operating system that runs from within the CD, without requiring any installation on the hard drive. This allowed to prepare a good execution environment for all the systems presented at the summer school: MATITA, COQ, Isabelle, Mizar, Agda and Epigram.

4.4.1 The database issue

The solution to the database related issue comes from the observation that, although the WHELP technology was developed on an enormous data set, the way it used in MATITA does not need all this dataset. MATITA has its own standard library, and even if it can use COQ objects, it is unlikely that the user interested in installing MATITA wants to install also the XML exportation of all COQ contributions (more than 1,5 gigabytes). Moreover, the architecture of MATITA was designed to be network transparent [80]: a so called getter components allows to fetch objects from remote hosts as if they were local. The existence of an embeddable and efficient SQL database[3] completes the picture.

As we mentioned before the DBMS is used to store also metadata of objects created by the user. This metadata are much less stable than the ones that are in the standard library, since the are frequently deleted or modified. This suggests the first distinction between the metadata set the user manipulates and the dataset

[1]`http://typessummerschool07.cs.unibo.it/`

[2]`http://mowgli.cs.unibo.it/~tassi/types/itplive/`

[3]`http://www.sqlite.org`

that composes the library officially distributed with the system. Both metadata has to be stored in a database, but they may be different. Moreover, since DBMS like MySQL accept connections through the network, they don't even need to be placed on the same host. Anyway, for performance and availability reasons, the standard library should be of quick access and available even if the user is working offline. After doing all these consideration we reworked the database related subsystem of MATITA as follows.

Metadata set — front-ends

MATITA distinguishes three kinds of metadata set. The one related to the user development where metadata related to objects defined by the user are stored. The library metadata set contains the metadata relative to the objects belonging to the standard library of the system; they are not changed by the user but will be heavily used. The third, and last, metadata set is the legacy one. It is again a read-only data set, unlikely to be heavily used by the common user but of interest for consultation.

DBMS — back-ends

MATITA supports two different, interchangeable, DBMS back-ends. The original MySQL, that can work across a network link, and the lightweight Sqlite. To make the latter work with MATITA effectively we had to enhance the OCaml bindings[4], allowing the declaration of user defined predicates (mainly for regular expression matching) to fill the gap between the SQL dialects spoken by the two different DBMS.

The database subsystem

Currently MATITA supports one instance of every kind of front-end that can be attached to any kind of back-end. The usual configuration is to keep both the user

[4]`http://www.ocaml.info/home/ocaml_sources.html`

and library metadata set local, handled by the configuration-pain free Sqlite back-end. The third metadata set, can be optionally activated, making the back-end point to the MySQL server properly configured available at the University of Bologna. If the remote metadata set is activated, the getter has also to be instructed where to find the XML representation of the objects whose metadata is available remotely.

Before this substantial re-design of the database subsystem of Matita all SQL queries were performed through a single DBMS on a single metadata set. Queries have been rewritten using only the intersection of the SQL dialects spoken by the two supported DBMS. Moreover all search queries are now performed on all active metadata set and the union is returned.

Picture 4.4 shows the internal architecture of the database subsystem when the default configuration is used. Dotted lines are related to the configuration file.

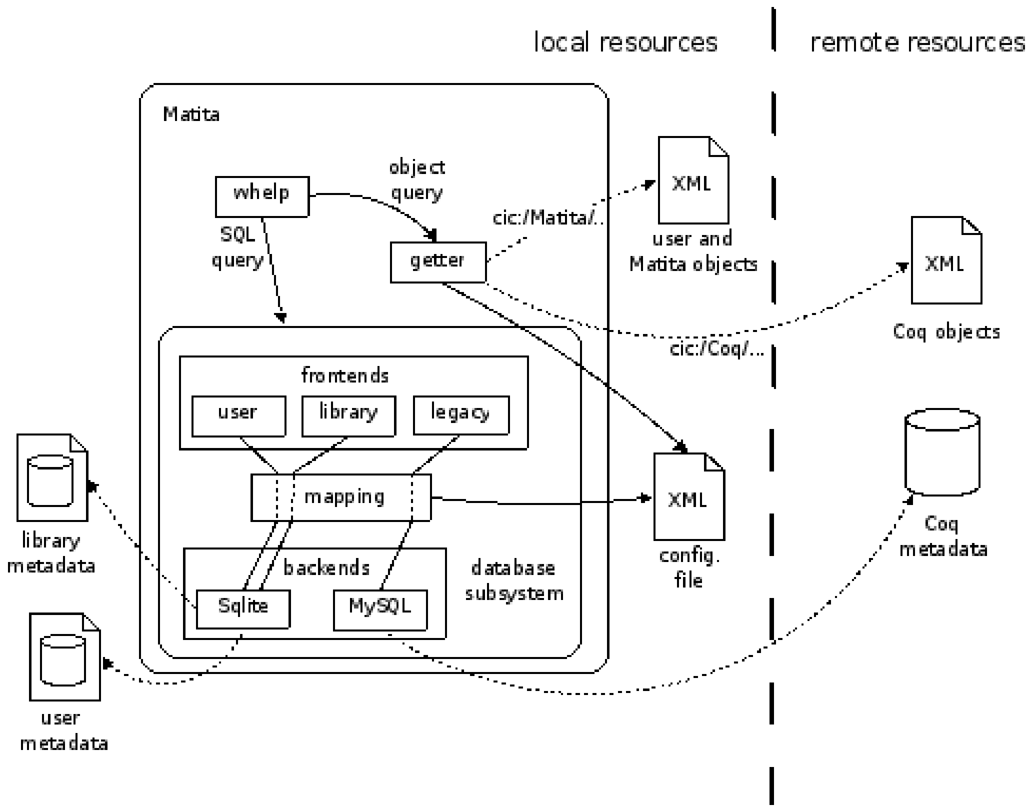


Figure 4.4: Database subsystem with default configuration

The main point of this design is to allow MATITA to still benefit from fast and scalable DBMS while reducing the effort needed by a user to perform a simple installation.

Limitation of this approach are that the metadata set activated as front-ends have to be disjoint. This limitation could be partially avoided using a priority policy, but in practice the simpler implemented approach of allowing only disjoint dataset proved to be effective and sufficient to most common needs.

4.4.2 A live CD for interactive theorem provers

ITPLive! is a live CD providing a desktop environment with many interactive (and some automatic) theorem provers already installed and ready to be used.

Live CDs are nowadays widespread medium for quick software evaluation, usually shipped with magazines to let the reader try a software product without the burden of installing it.

Interactive theorem provers are often hard to install, sometimes requiring uncommon dependencies to be satisfied. Moreover they usually need to be properly configured. We were also involved in the organisation of the 2007 Types Summer School, with around 80 participants. Writing an how-to and assisting them in the installation on their own computer would have been a huge amount of work. We thought that the live CD approach was suitable for interactive theorem provers.

We chose to base the live CD on the Debian GNU/Linux, because all interactive theorem prover works under a Unix operating system and because we master the details of that Linux distribution.

The technology behind a live CD can be dived into two parts: the one in kernel space and the one in user space. Both are described in the following sections.

The technology for a live CD

The first technology, continuously improved in the last few years after the success of the Knoppix distribution, accounts to an highly compressed file system and to an extremely flexible file system indirection layer. Cause the limited amount of space on a CD media, it is necessary to heavily compress the content of the file system to make all necessary software fit. In addition to that, CD are read only media and a live CD has to install nothing on the hard drive. Applications are usually developed having in mind the most common use case, where a writable file system is available. Thus most applications create temporary files, or populate the users home directory with configuration files or even use the hard drive as a cache for remote contents.

The technology used to solve the compression related issue is *squashfs*[5], a kernel module and user space tool to create and mount compressed file system images. It uses a slightly modified version of the well known Ziv-Lempel LZ77 [97] algorithm that 30 years after its discovery still represent a good compromise between (de)compression performances and compression ratio. It performed incredibly well with the so huge Mizar library, composed of around 300 mega bytes of text files. The uncompressed size of the whole file system, comprising the operating system, a graphical user interface and all the provers amounts to more than two gigabytes and was compressed down to around six hundred megabytes.

To solve the read only media problem the extremely powerful file system indirection layer *unionfs*[6] was adopted. It allows to merge a writable and a read only filesystem such that every modification to the read only one is recorded in the writable one. Modern computers have huge main memory, and few megabytes can be safely used to create a ramdisk. This writable virtual drive can be formatted with a regular file system and be used together with the read only one provided by the CD thanks to *unionfs*. All application can thus work without modification, they are free to write on the CD media as they used to do on the hard drive: all their modification are stored in main memory.

[5]`http://squashfs.sourceforge.net/`
[6]`http://www.unionfs.org/`

The tools for a live CD

The combination of *unionfs* and *squashfs* is enough to make live CD possible, but without an handy tool to create the file system image it would be an incredibly time consuming activity.

The *debian-live*[7] project, although still on it early stage of development, was adopted to drive the generation of the file system image to put on the live CD. We contributed with a quite huge number of bug reports and patches to the project, that constantly improved and fixed the utility.

What *debian-live* allows to do is to specify a list of Debian packages that will be available when running the live CD. It also handles a customised boot sequence (thanks to the *live-initramfs* utility) more suitable for the live CD. It makes it easy to add on top of that automatically generated file system image ad-hoc software that is not available as a Debian package and to customise the user desktop adding startup icons for the installed provers.

Having that utility allowed us to generate many tentative live CD in batch mode, quickly improving the quality of the product.

The contents of ITPLive!

The version of the live CD given to the 2007 Types summer School attenders can be download in the website of the school.

It provides recent versions of MATITA, COQ, Isabelle, Mizar, Agda and Epigram. In addition to these interactive theorem provers it provides the why [42] and krakatoa [61] software verification tools. It also ships some automatic provers why can use, like simplify [37], ergo [28] and yices [39].

After the database related issue of MATITA was solved reworking the database subsystem, we made a Debian package for the tool that is used to install it on the live CD. Since some attenders the conference were using the Ubuntu Linux distribution, that shares with Debian the same package management system, they successfully

[7]`http://debian-live.alioth.debian.org/`

installed the package on their laptops. Ubuntu is a really widespread distribution for personal desktops/laptops, we thus look forward in making the MATITA Debian package part of the official Debian distribution. The whole official Debian distribution is available as part of the Universe repository to all Ubuntu users.

Chapter 5

Coercive subtyping in Matita

The refiner is a fundamental component in an interactive theorem prover like MATITA. Looking back at the data driven architecture description of Section 4.2, the refiner handles incomplete terms. This kind of data is usually the result of the user input, that must be validated possibly adjusting it inferring implicit piece of information.

A refiner has always been part of MATITA, even before the user interface was written. It was developed during the Mowgli project, as part of the algorithm that efficiently disambiguates mathematical formulae [82, 83] where constant names are overloaded.

Even if the refiner was already working [80], no support for implicit coercions was available. We developed it as part of our PHD, with two main motivations:

- implicit notation support
- formalisation of algebraic structures

When we started developing coercion support in the refiner, MATITA was equipped with no notational support, not even infix operators. However, the implementation of explicit notational support was planned, many common practices in pen & paper do not fit in this category and need implicit notation. The less interesting one, but still emblematic of the implicitness of some notations is for example the formula $3x+5 = 0$ that silently misses the $*$ operation on 3 and x. A more interesting example in which an implicit notation is usually adopted is when an object has multiple aspects, like an abelian group that is also a set of objects or another algebraic structure like a monoid. Many of these interesting use cases fall in the subtyping category, that may not be supported by the logic framework implemented by the interactive theorem prover. CIC, the calculus implemented in MATITA is an example of such framework. The most commonly used encoding of algebraic structures in a type theory like CIC is to use inductive telescopes (i.e. generalised Σ-types) to model structures and implicit coercions to model the subtyping relation.

Although these are the main motivations of our work, the flexible handling of metavariables MATITA employs allowed us to further extend the coercion mechanism. It is for example possible to declare subset coercions, like nat_of_Z : $\forall$ n:Z.n$\geq$0.nat, that, when inserted around a term n of type Z, opens the conjecture n$\geq$0. To fully exploit this feature some additional operations have to be implemented (like coercion propagation under term constructors see 5.4.1), allowing to declare coercions to Σ-types that may leave the propositional part to be proved. This implements the core functionality of the Russell [87] system available in COQ or the predicate subtyping feature of PVS [85] that allows to specify functional programs in a very intuitive way that also plays nice with code extraction.

This chapter is structured as follows: Section 5.1 introduces various aspects coercions, from their role in the encoding of mathematical structure in type theory to their formal presentation and implementation. Section 5.2 describes the design choice we made and details the implementation of coercions in the MATITA refiner. An extended rework of the paper “Working with Mathematical Structures in Type Theory” [27] accepted for publication in the TYPES 2007 post proceedings is reported in Section 5.3. It gives a technical presentation of the issues of coding mathematical structures with a notion of subtyping in type theory, explaining the solutions adopted in MATITA. In the last Section 5.4 an overview of some features, like subset coercions to Σ-types, is given.

5.1 The many faces of coercions

Coercions have been widely studied in the literature from very heterogeneous points of view. The theoretical approach lead to nice results defining correspondences between languages with subtyping and languages that simulate that feature through the application of coercions. Implementation of coercions in interactive theorem provers usually deeply differs from the theoretical presentation, mostly for performance reasons. Restrictions are usually performed on the category of functions allowed to be coercions and the typing rules that insert coercions have stronger premises than the ones usually presented in theoretical studies. Applicative studies concerning the formalisation of algebraic structures in type theory used them as an handy gadget to mimic subtyping between algebraic structures.

Our work concentrated on the implementation of the coercions mechanism in the interactive theorem prover MATITA, tuning this mechanism to ease the formalisation of a hierarchy of algebraic structures.

Here we give a brief introduction to all the three aspects of coercions.

5.1.1 Theoretical properties

Theoretical aspects of coercions have been extensively studied by Luo [57, 58], Luo and Soloviev [59, 55], Barthe [11] and Chen [22, 23].

Luo and Soloviev base the study on a typed version of Martin-Löf Logical Framework (see [57] for the full set of rules) and they relate two extensions of the framework, the first obtained equipping the theory with a subtyping notion, and the second obtained adding coercions as mean of an abbreviational mechanism. They define the notion of coherence of the coercion graph declarations and, under such assumption, they prove the conservativity of the latter extension over the former. Luo and Soloviev [59] generalise the work done in [57] considering not only simple coercions, but also the parametrised and dependent case. An example of parametrised coercion is the one that maps vectors (of any given length) to lists: $\mathrm{V2L} : \forall\,\mathrm{n}.\mathrm{Vector}\ \mathrm{n} \rightarrow\ \mathrm{List}$ An example of the more interesting dependent coercion is the dual one, mapping lists to vectors of a length that depends on the input: $\mathrm{L2V} : \forall \mathrm{l} :\ \mathrm{List}.\ \mathrm{Vector}\ (\mathrm{length}\ \mathrm{l})$

The coherence condition, as expressed in [59], states that two coercions from the same target to equal types are equal:

Coherence condition

$$\begin{array}{l} \Gamma \vdash x : A \xrightarrow{c} B[x] : \mathit{Type} \quad \Gamma \vdash x : A \xrightarrow{c'} B'[x] : \mathit{Type} \\ \Gamma \vdash a : A \quad \Gamma, a : A \vdash B[a] = B'[a] \\ \Rightarrow \Gamma \vdash c\ a = c'\ a : B \end{array}$$

Judgements of the form $\Gamma \vdash x : A \xrightarrow{c} B[x] : \mathit{Type}$ are used to state the declaration of a dependent coercion from A to B. Although in [57] Luo does not consider coercions between equal types, in the subsequent paper [59] he does and in conjunction with the coherence condition he is able to deduce that if $\Gamma \vdash A \xrightarrow{c} A : Type$ then $\Gamma \vdash c = \lambda x : A.x : A \to A$.

The conservativity property intuitively states that each derivation in the logical framework extended with coercive subtyping that does not contain coercive applications can be mapped to a derivation in the original logical framework. This result is established in [59] for the most complicated case of dependent coercions.

Barthe [11] works inside a Pure Type System equipped with $\beta\eta$-equivalence and studies coercions as a relation between a system with a fully explicit syntax and one with implicit syntax. The system equipped with implicit syntax has an extra rule to introduce coercions into a set Δ that annotates every type judgement rule. The side conditions for that rule are shown to be sufficient to prove the conservativity of the system with implicit syntax over the other one. A notion of ϵ-reduction, that makes coercions explicit, is used to formulate the conservativity and coherence statements: given a judgement $\Gamma \vdash_\Delta M : T$ in the system with implicit syntax, $\epsilon_\Delta(\Gamma) \vdash \epsilon_\Delta(M) : \epsilon_\Delta(T)$ is provable in the explicit syntax system. Moreover if M_1 and M_2 are ϵ-normal forms of M, $M_1 =_\beta M_2$. Restrictions on the coercion declaration rules are similar to the ones made by Luo and Soloviev, different coercive paths between the same types have to be β-convertible. Barthe also does not investigate transitivity in details, refusing to add to the coercion set a new coercion from vectors over B to a lists over B when a coercion from A to B and a coercion from vectors over A and lists over A are declared.

Chen [22, 23] investigates the transitivity related problem and develops an extension of the calculus of constructions where multiple, different, paths of implicit coercions between equal types are allowed. He develops a coercion inference algorithm, using the calculation of the least upper bound [6] to chose the coercion to insert. This allows him to prove that the algorithm produces the minimal well typed coercive extension of an ill typed term. His rule for application always inserts coercions, and the least upper bound calculation allows him to assume that they are identities (always declared as coercions) if the types are convertible.

All these works give a clean formal presentation of type systems (or type system frameworks) equipped with a syntax for implicit coercions, and prove important results over them. Although the requirements they put on coercions are too strong for practical usage. The coherence condition is in general undecidable in the setting of Luo and Soloviev [59]. Moreover a coercion $c : A \to B$ is chosen to cast a term of type A' when A and A' are $\beta\eta$-convertible, that can lead in general to extremely expensive conversion tests during type checking (or the coercion inference phase if type checking is performed on the system with explicit syntax). The implementations made by Bailey and Saibi for Lego and COQ described in the next section tackle this problem adding some restrictions to the coercion application rules.

5.1.2 Implementations

Two rather similar implementations of coercive subtyping have been made by Saibi [84] for COQ and by Bailey [7, 8] for Lego. The first drift from the theoretical treatment of Luo and Soloviev [59] is that they implement coercive subtyping in a specific type theory and not as a generic abbreviation mechanism in a logical framework. They also adopt stronger limitations than Barthe [11] and Chen [23] since they perform only syntactic (by name) matching of types. The main reason for such a limitation is that searching a possibly large graph of coercions using convertibility is too expensive to be adopted in practice. In particular they identify the source and target of coercion with the head constant (usually a type constructor) name, thus the coercion graph is searched by means of a cheap label comparison. As spotted by Luo [57] coherence checking is unfeasible in practice, and Saibi circumvents the problem disallowing the declaration of multiple paths between the same labels. Saibi make additional restrictions on the types coercions can have called uniform inheritance. Let C and D be type constructors with respectively n and m parameters. A coercion from C to D can be declared if it has type of the form

$$\forall x_1 \colon A_1 \ldots x_n \colon A_n.\ c : C\ x_1 \ldots x_n.\ D\ u_1 \ldots u_m$$

This allows him to easily infer all the parameters x_i just looking at the actual type $\mathrm{C}\ \mathrm{v}_1 \cdots \mathrm{v}_n$ of the coerced term. Bailey allows C not to be uniform on its arguments and uses the type inference mechanism of Lego to try to infer the parameters of c (that always succeeds if the type of the coercion respects the uniform inheritance condition).

Both Saibi and Bailey identify three different kind of coercions, the most intuitive one is between types, one from types to sorts and another one from types to the product space. The rules they use to insert coercions in [84, 8] are very similar. Rules as formulated by Saibi follow. Δ is a graph of coercions, Γ a regular context of variables declarations. The $\Rightarrow$ symbol is used to represent the refinement function. s_i ranges over allowed sorts and $\Delta^D(t : C)$ is the coercion lookup from C to D where D may be Π or SORT to look for coercions to dependent products or sorts. $\mathcal{R}$ is the function calculating the new sort of a product (note that in CIC all quantifications are allowed and the PTS is functional thus in the general case $\mathcal{R}$ return the second input, unless both inputs are $\mathrm{Type}_{\mathrm{i/j}}$ where $\mathrm{Type}_{\mathrm{max(i,j)}}$ is returned). The first rule defines the relation $\overset{s}{\Rightarrow}$ that ensures that the right hand side is a sort.

BS-sort

$$\frac{\Delta, \Gamma \vdash A \Rightarrow A' : T}{\Delta, \Gamma \vdash A \overset{s}{\Rightarrow} \Delta^{\mathrm{SORT}}(A' : T) : Type}$$

BS-prod

$$\frac{\Delta, \Gamma \vdash T \overset{s}{\Rightarrow} T' : s_1 \quad \Delta, \Gamma; x : T' \vdash U \overset{s}{\Rightarrow} U' : s_2}{\Delta, \Gamma \vdash \Pi x : T.U \Rightarrow \Pi x : T'.U' : \mathcal{R}(s_1, s_2)}$$

BS-lam

$$\frac{\Delta, \Gamma \vdash T \overset{s}{\Rightarrow} T' : s \quad \Delta, \Gamma; x : T' \vdash t \Rightarrow t' : U \quad \Delta, \Gamma \vdash \Pi x : T'.U \overset{s}{\Rightarrow} V : s}{\Delta, \Gamma \vdash \lambda x : T.t \Rightarrow \lambda x : T'.t' : V}$$

BS-app

$$\frac{\Delta, \Gamma \vdash t \Rightarrow t' : T \quad \Delta, \Gamma \vdash u \Rightarrow u' : U \quad \Delta^{\Pi}(t' : T) = \Pi x : A.B}{\Delta, \Gamma \vdash (t\ u) \Rightarrow (\Delta^{\Pi}(t' : T)\ \Delta^{A}(u' : U)) : B[\Delta^{A}(u' : U)/x]}$$

The coercion lookup function $\Delta^D(t : C)$ returns the term t unchanged if D and C are convertible, while returns c $\mathrm{p}_1 \cdots \mathrm{p}_n \mathrm{t}$ where p_i is a parameter of the coercion c.

Note that the uniform inheritance condition enforced in the implementation by Saibi gives a complete algorithm to obtain each p_i, while the implementation of Bailey uses the refinement procedure to infer them.

Callaghan [20] implements in the experimental interactive theorem prover Plastic [21] a more flexible flavour of coercions, dropping the limitation of syntactic type matching for coercion search. He drops this requirement to allow a deeper experimentation of coercions, considering this the main aim of Plastic.

An interesting feature of its implementation is the computation of an approximation of the transitive closure of the coercion graph. Consider the case in which c is declared as a coercion. The following set of coercions (when they are well typed) is added to Δ.

$$\{c_i \circ c \circ c_j | c_i \in \Delta \wedge c_j \in \Delta\} \cup \{c \circ c_j | c_j \in \Delta\} \cup \{c_i \circ c | c_i \in \Delta\}$$

In general this algorithm computes only an approximation of the closure of the graph, that could be infinite. Coercions not generated automatically can always be declared by hand. The lookup function does not try to generate new coercions on the fly.

An interesting mechanism to ameliorate this last limitation is a limited form of transitivity, that allows to declare coercions parametric over coercions like

$$\forall \mathrm{A,A',B}.\forall \bar{f}\mathrm{: A} \rightarrow \mathrm{A'.\ A{\times}B} \rightarrow \mathrm{A'{\times}B}$$

where $\bar{\cdot}$ is used to state that f must be a coercion. In this way he can generate some coercions on the fly in a lazy fashion.

All these implementations provide a useful tool for formalising structures with inheritance in a type theory without subtyping. In our experience, at least one additional feature is needed to effectively formalise structure with inheritance: the possibility of sharing substructures.

In the next section we describe Pollack's work on the encoding of mathematical structures with manifest fields (thus possibly shared) in type theory. He obtains extremely good results using induction-recursion.

5.1.3 Encoding mathematical structures

In [74] Pollack shows how to interpret dependently typed records with and without manifest fields in a simpler type theory having only primitive Σ-types and primitive Ψ-types. A Σ-type $(\Sigma\, x{:}T.P\ x)$ is inhabited by heavily typed couples $\langle w,p\rangle_{T,P}$ where w is an inhabitant of the type T and p is an inhabitant of (P w). The heavy type annotation is required for type inference. A Ψ-type $(\Psi x{:}T.p)$ is inhabited by heavily typed singletons $\langle w\rangle_{T,P,p}$ where w is an inhabitant of the type T and p is a function mapping x of type T to a value of type (P x). The intuitive idea is that $\langle w,\ p[w]\rangle_{T,P}$ and $\langle w\rangle_{T,P,\lambda x:T.p[x]}$ should represent the same couple, where in the first case the value of the second component is opaque, while in the second case it is made manifest (as a function of the first component). However, the two representations actually *are* different and morally equivalent inhabitants of the two types are not convertible, against intuition.

We will denote by .1 and .2 the first and second projection of a Σ/Ψ-type.

The syntax "$\Sigma\, x{:}T.P\ x$ **with** $.2 = t[.1]$" can now be understood as syntactic sugar for "$\Psi x{:}T.t[x]$". The illusion is completed by declaring a coercion from $\Psi x{:}T.p$ to $\Sigma\, x{:}T.P\ x$ so that $\langle w\rangle_{T,P,p}$ is automatically mapped to $\langle w,\ p\ w\rangle_{T,P}$ when required.

Most common mathematical structure are records with more than two fields. Pollack explains that such a structure can be understood as a sequence of left-associating[1] nested heavily typed pairs/singletons. For instance, the record $r\ \equiv\langle \mathrm{nat},\ \mathrm{list\ \ nat},\ @\rangle_R$ of type $R := \{C : \mathrm{Type};\ T := \mathrm{list}\ \ C;\ \mathrm{app}{:}\ T\ \rightarrow\ T\ \rightarrow\ T\}$, where the second field is explicit, is represented as

$$
\begin{aligned}
&T_0 \equiv \Sigma C : \mathrm{Unit}.\ \mathrm{Type}\\
&r_0 \equiv \langle\, (),\ \mathrm{Type}\rangle_{\mathrm{Unit},\ \lambda C:\mathrm{Unit}.\mathrm{Type}}\\
&T_1 \equiv \Psi y{:}T_0.\ \mathrm{list}\ \ y.1\\
&r_1 \equiv \langle r_0\rangle_{T_0,\ \lambda x:T_0.\mathrm{Type}_1,\ \lambda y:T_0.\mathrm{list}\ y.1}\\
&r\ \equiv \langle r_1,\ @\rangle_{T_1,\ \lambda x:T_1.\ x.2\rightarrow x.2\rightarrow x.2}
\end{aligned}
$$

[1]In the same paper he also proposes to represent a record type with a right-associating sequence of Σ/Φ types, where a Φ type looks like a Ψ type, but makes it first fields manifest. However, in Sect. 5.2.2 he also argues for the left-associating solution.

of type Σx:(Ψy:(ΣC: Unit. Type). list y.1).x.2 → x.2 → x.2.
However, the deep heavy type annotations are actually useless and make the term extremely large and its type checking inefficient. The interpretation of **with** also becomes more complex, since the nested Σ/Ψ types must be recursively traversed to compute the new type.

In [74], Pollack shows that dependently typed records with uniform field projections and **with** can be implemented in a type theory extended with inductive types and the induction-recursion principle [40]. However, induction-recursion is not implemented in most proof assistants. In Section 5.3 we propose a solution in a simpler framework where we only have primitive records (or even simply primitive telescopes), but no induction recursion.

5.2 Implementation of coercions in Matita

MATITA is a relatively small and young system. The type inference subsystem (that we will call refiner) has always been extremely flexible allowing an uniform treatment of metavariables (see [80]). The design choices we made when we implemented coercions have been inspired by the work of Saibi [84], Bailey [7] and Callaghan [20]. We followed the approach of Saibi and Bailey regarding the syntactic match of coercion types, but completely relaxing all limitations like the uniform inheritance condition. Moreover we adopted the same approach to the (partial) transitive closure of the coercions graph that is used in [20]. We also allow multiple incoherent paths between the same types. This choice allows to better handle parametric inductive types, that may differ only by parameters and are collapsed to the same node by the coarse syntactic (by name) comparison. Some examples of coercions allowed thanks to these relaxed constraints are the following:

```
definition z2nat : ∀x : Z. x ≥ 0 → nat.
definition l2lnnil : ∀l : list . length l > 0 → { l : list , l ≠ nil}.
definition morphonlist : morph A B → list A → list B.
definition morphontree : morph C D →tree C → tree D.
```

The former example allows to cast integers to natural number provided a proof that the integer is grater than zero. It clearly breaks the uniform inheritance condition. The second example uses the same technique to inject an object in a sigma type letting to be proved, as a side condition, that the object validates the predicate identifying the sigma type.

The latter two examples are both from the syntactic label morph to the special class of products, but are clearly incoherent. If C and A are not convertible (or D and B) then these coercions clearly relates different types, but are usually rejected by systems not allowing multiple incoherent paths between the same labels. Although it seems reasonable to allow both coercions to be declared, since is A and C are not convertible, only one of the two coercions can actually be applied. What has been observed in the every day usage is that the coarse label approximation of types can lead to multiple results when a search is performed, but the subsequent unification step always makes the choice of the coercion to apply deterministic (i.e. only one coercion application is effectively well typed). Clearly the user can construct an example in which this does not always happen, but we found no non-artificial examples that suffers from an arbitrary choice of the inserted coercion.

We may also declare coercions like the following one, that allows to always use a list disregarding the type of its elements, letting the user to later provide a function mapping the two types.

```
definition maplist : ∀A,B. ∀f : A → B. list A → list B.
```

In general this coercion is too weak, since no requirements on f are made. Requiring f to be a bijection, can make this coercion really handy when working up to type isomorphisms.

5.2.1 Syntax and notation

Here we give a syntax for CIC terms and the notation we will use to describe the refinement algorithm. In CIC, proofs and types live in the same syntactic category. Terms are described in Table 5.2.1, coinductive types/fixpoints are not presented

t	$::=$	x	identifiers
	$\mid$	c	constants
	$\mid$	I	inductive types
	$\mid$	k	inductive constructors
	$\mid$	$Prop \mid Type(j)$	sorts
	$\mid$	$t\ t$	application
	$\mid$	$\lambda x : t.t$	λ-abstraction
	$\mid$	let $x := t$ in t	local definitions
	$\mid$	$\Pi x : t.t$	dependent product
	$\mid$	match t in I return t [	case analysis
		$k_1\ \overline{x_1} \Rightarrow t \mid \ldots \mid k_n\ \overline{x_n} \Rightarrow t$	
		]	
	$\mid$	letrec $f_{n_1}(x_1 : t) \cdots (x_{k_{n,1}} : t) : t$ on $l_{n_1} := t$ and $\ldots$	
		and $f_{n_m}(x_1 : t) \cdots (x_{k_{n,m}} : t) : t$ on $l_{n_m} := t$ in f_{jj}	recursive definitions
	$\mid$	$?$	implicit arguments
	$\mid$	$?j[t\ ;\ \ldots\ ;\ t]$	metavariable occurrence

Table 5.1: CIC terms syntax

since they will play no role in this presentation. We reserve j, l, m, n for integers, t, M, N for terms, T, S, Q for types (i.e. terms used as types), x, y, z for variables, i will be used as the index of iterations. I is reserved for inductive types.

In Table 5.2.1 l_{n_m} is lesser or equal $k_{n,m}$ and represents the argument of the recursive definition that is expected to syntactically decrease in every recursive call. $(y_1 : t) \cdots (y_{k_{n,m}} : t)$ is a possible empty sequence of variables bound in the body of the function. As usual, $\Pi x : T_1.T_2$ is abbreviated in $T_1 \to T_2$ when x is not a free variable in T_2. The inductive type I in the pattern matching constructor is redundant, since distinct inductive types have distinct constructors; it is given for the sake of readability. The term introduced with the return keyword will be used to obtain the return type of the pattern matching. Variables $\overline{x_i}$ are abstracted in the right hand side terms of $\Rightarrow$.

Implicit arguments, written ? are placeholders for missing (untyped) terms and occur linearly. Metavariable occurrence, represented with $?j[t\ ;\ \ldots\ ;\ t]$, are missing typed terms living a specific context and are equipped with a local substitution (that may be omitted if not interesting). The CIC calculus extended with metavariables has been studied in [65] and the flavour of metavariables implemented in Matita is described in [80].

The syntax presented in Table 5.2.1 is extremely similar to the concrete syntax of Matita, that simply relaxes all typing informations (making them optional and replacing them with implicit arguments, see Section 4.2). The return type of the pattern matching construct can be omitted as well. Notational facilities, although supported by Matita (see [72]) are not considered here.

5.2.2 Preliminary definitions

To describe the refinement algorithm we need to define some structures and operations over them. Following the naming convention of [80, 89] we define what a problem $\mathcal{P}$ is.

Definition 5.1 (Proof problem) *A* proof problem $\mathcal{P}$ *is a finite list of typing judgement of the form* $\Gamma_{?j} \vdash ?j : T_{?j}$ *where for each metavariable that occurs free in* $\mathcal{P}$ *there exists a corresponding sequent in* $\mathcal{P}$*.*

A proof problem, as well as a CIC term, can refer to constants, that usually live in an environment that decorates every typing rule. In the following presentation we consider a global well formed environment E where all constants and inductive types are associated with their types. No typing rules will modify this environment. The implementation of this environment in Matita is in fact lazy, and constants are added only when it is strictly necessary. Related theory is studied in [24, 80], and plays no role in this presentation. We thus omit the environment E almost everywhere.

Proof problems do not only declare missing proofs (i.e. not all $T_{?j}$ have sort *Prop*) but also missing terms. For example the coercion maplist shown before, when applied, will generate a proof problem that can be closed by providing an inhabitant of A $\rightarrow$ B that has sort *Type*.

Definition 5.2 (Metavariables of term/context ($\mathcal{M}$)) *Given a term t, $\mathcal{M}(t)$ is the set of metavariables occurring in t. Given a context Γ, $\mathcal{M}(\Gamma)$ is the set of metavariables occurring in Γ.*

The function $\mathcal{M}$ is at the base of the order relation defined between metavariables.

Definition 5.3 (Metavariables order relation ($\ll_{\mathcal{P}}$)) *Let $\mathcal{P}$ be a proof problem. Let $<_{\mathcal{P}}$ be the relation defined as: $?n_1 <?n_2$ iff $?n_1 \in \mathcal{M}(\Gamma_{?n_2}) \cup \mathcal{M}(T_{?n_2})$. Let $\ll_{\mathcal{P}}$ be the transitive closure of $<_{\mathcal{P}}$.*

Definition 5.4 (Valid proof problem) *A proof problem $\mathcal{P}$ is a* valid proof problem *if and only if $\ll_{\mathcal{P}}$ is a strict partial order (or, equivalently, if and only if $\ll_{\mathcal{P}}$ is an irreflexive relation).*

The intuition behind $\ll_{\mathcal{P}}$ is that the smallest $?j$ (or one of them since there may be more than one) does not depend on any other metavariable (e.g. $\mathcal{M}(\Gamma_{?j}) = \emptyset$ and $\mathcal{M}(T_{?j}) = \emptyset$ where $\Gamma_{?j} \vdash ?j : T_{?j} \in \mathcal{P}$). Thus instantiating every minimal $?j$ with a metavariable free term will give a new $\mathcal{P}$ in which there is at least one $?j$ not depending on any other metavariable (or $\mathcal{P}$ is empty).

Since we do not consider environments E in our typing rule the definition of *well-formed proof problem* is slightly simpler then the one given in [80].

Definition 5.5 (Well formed proof problem) *A valid proof problem $\mathcal{P}$ is a* well-formed proof problem *if an only if for all $(\Gamma_{?j} \vdash ?j : T_{?j}) \in \mathcal{P}$ we have $\mathcal{P}, \Gamma_{?j} \vdash T_{?j} : s$ and s is a sort. The judgement $\mathcal{WF}(\mathcal{P})$ states that the valid proof problem $\mathcal{P}$ is well-formed.*

The typing judgement used in the previous definition is an extension of the regular typing judgement.

Definition 5.6 (Typing judgement) *Given a term t not containing implicit arguments and given a well formed proof problem $\mathcal{P}$ that contains all metavariables in t we write*

$$\mathcal{P}, \Gamma \vdash t : T$$

to state that t is well typed. This typing judgement is an extension of the standard one for CIC where:

- *Substitution is extended with the following rule*

$$?i[t_1, \dots, t_n]\sigma = ?i[t_1\sigma, \dots, t_n\sigma]$$

- *Convertibility relation is enlarged allowing reduction to be performed inside metavariables explicit substitution.*

$$\frac{\Gamma \vdash t_i \downarrow t'_i \qquad\qquad i \in \{1 \dots n\}}{\Gamma \vdash ?j[t_1\ ;\ \dots\ ;\ t_n] \downarrow ?j[t'_1\ ;\ \dots\ ;\ t'_n]}$$

- *The following typing rules are added*

$$\frac{\begin{array}{c} y_1 : T_1\ ;\ \dots\ ;\ y_n : T_n \vdash ?j : T_{?j} \in \mathcal{P} \\ \Gamma \vdash t_i : T_i[y_1/t_1\ ;\ \dots\ ;\ y_{i-1}/t_{i-1}] \quad i \in \{1 \dots n\} \end{array}}{\Gamma \vdash \mathcal{WF}(?j, [t_1, \dots, t_n])}$$

$$\frac{(y_1 : T_1\ ;\ \dots\ ;\ y_n : T_n \vdash ?j : T_{?j}) \in \mathcal{P} \quad \Gamma \vdash \mathcal{WF}(?j, [t_1, \dots, t_n])}{\Gamma \vdash ?j[t_1, \dots, t_n] : T_{?j}[y_1/t_1\ ;\ \dots\ ;\ y_n/t_n]}$$

We used $\mathcal{WF}(?j, [t_1, \dots, t_n])$ to state that a metavariable occurrence is well formed in Γ.

Definition 5.7 ((Valid) Instantiation) *An instantiation ι is a function from metavariables to terms. It is* valid *if $Dom(\iota) \cap \mathcal{M}(\iota(Dom(\iota))) = \emptyset$*

The instantiation operation is recursively performed over a term. It behaves as the identity in every case except when a metavariable belonging to its domain is encountered.

$$\iota(?j[t_1; \ldots; t_n]) = \iota(?j)[\iota(t_1); \ldots; \iota(t_n)]$$

Metavariable instantiations seen as functions is an elegant theoretical approach, but using closure to represent substitutions is not efficient. Interactive theorem prover usually implement a mechanism to pack substitutions together, allowing to postpone their application as much as possible.

Definition 5.8 (Metavariable substitution environment) *A metavariable substitution environment Σ (called simply substitution when non ambiguous) is a list of couples metavariable-term.*

$$\Sigma = [?1 := t_1; \ldots; ?n := t_n]$$

The operation $\Sigma(t)$ is defined as $\iota_n(\ldots \iota_1(t) \ldots)$ where ι_i is the function mapping $?i$ to t_i. It is extended to contexts Γ in the following way

$$\Sigma(x_1 : T_1 \ ; \ \ldots \ ; \ x_n : T_n) = x_1 : \Sigma(T_1) \ ; \ \ldots \ ; \ x_n : \Sigma(T_n)$$

Valid substitutions do apply also to well formed proof problems. Consider that what the user sees of an ongoing proof is exactly the proof problem, thus all instantiations collected so far have to be made explicit.

$$\Sigma(\Gamma_{?j} \vdash ?j : T_{?j}) = \Sigma(\Gamma) \vdash ?j : \Sigma(T_{?j}) \qquad \text{(for each } ?j \in \mathcal{P})$$

What in Section 4.2 and [2] is called partially specified term falls in the syntactic category of Table 5.2.1.

Definition 5.9 (Raw term) *A* raw term *is a term possibly containing implicit arguments.*

Raw terms are a subclass of partially specified terms (using the terminology introduced in Section 4.2. Raw terms are incomplete, potentially ill-typed. They are the result of the user input, where all the syntax allowed to be optional by the concrete syntax and omitted (like types) is replaced with implicit arguments. Notational conventions have already been processed, thus raw terms completely fall in the syntactic category described in Table 5.2.1.

The concrete syntax of MATITA allows to type in a metavariable and its explicit substitution directly, but is mainly used for debugging purposes. We can thus consider raw terms t where $\mathcal{M}(t) = \emptyset$.

Definition 5.10 (Refined term) *A* refined term *is a term t together with a, possibly empty, proof problem $\mathcal{P}$ and a substitution Σ such that it is well typed*

$$\Sigma(\mathcal{P}),\ [] \vdash \Sigma(t) : \Sigma(T)$$

.

Refined terms are the output of the refinement process and do not contain implicits (e.g. all implicits have been turned into metavariables typed in $\mathcal{P}$).

Definition 5.11 (Refiner) *The refiner is an algorithm taking in input a raw term and giving in output a refined term or raising an error. The refinement process is identified using the following notation*

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash t \overset{\mathcal{R}}{\leadsto} t' :\ T,\ \mathcal{P}',\ \Sigma'$$

where t' is the result of the refinement process and is well typed of type T in $\mathcal{P}'$, Σ' and Γ:

$$\Sigma'(\mathcal{P}),\ \Sigma'(\Gamma) \vdash \Sigma'(t') : \Sigma'(T)$$

Since refinement deals with terms containing flexible parts, conversion tests are replaced with unification tests. In a higher order and dependently typed calculus like CIC unification is in the general case undecidable. What is usually implemented in interactive theorem provers is an essentially fist order unification algorithm, handling only some simple higher order cases in an ad-hoc manner. A notable exception is Isabelle [54] that performs full second order unification using Huet's algorithm. The unification algorithm implemented in MATITA is not explained here in details, the interested reader can find more details in [80].

Definition 5.12 (Unification) *The process of unifying two terms is denoted with*

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash N \overset{?}{\equiv} M \ \overset{\mathcal{U}}{\leadsto}\ \mathcal{P}',\ \Sigma'$$

Unification performs only metavariables instantiations, and the resulting Σ' is such that $\Sigma'(N)$ is convertible with $\Sigma'(M)$ in context $\Sigma'(\Gamma)$ and proof problem $\Sigma'(\mathcal{P}')$. Coercions come to play when unification fails (i.e. an explicit cast is needed). The substitution application operation is seldom used explicitly, since all judgement take in input and give back a substitution. Another operation used in the operational presentation of rules in [80] is *whd*, performing weak head $\beta\zeta\iota$-reduction. The syntax presented in Table 5.2.1 does not include local definitions, thus ζ-reduction can not be performed. On the contrary ι-reduction, concerning fixpoint and pattern matching, is present and will be used. In this presentation head $\beta\iota$-reduction and substitution application are considered implicitly used when needed. For example when an unification of a type with a dependent product is performed, $\beta\iota$-reduction as well as metavariable instantiation is performed if needed.

Definition 5.13 (Explicit (optional) cast) *The explicit cast of a term M of type T_1 to the type T_2 is denoted by the following notation*

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash M : T_1 \overset{?}{\equiv} T_2 \ \overset{\mathcal{C}}{\leadsto}\ M',\ \mathcal{P}',\ \Sigma'$$

Explicit casts can modify a term inserting coercions when needed. The graph of coercions is considered part of a global state since refinement rules do not modify it. We will refer to it with Δ.

Definition 5.14 (Coercion lookup) *Coercion lookup is denoted as follows*

$$\mathcal{P},\ \Gamma \vdash T_1 \rightarrowtail T_2 \ \overset{\Delta}{\leadsto}\ k,\ c\ ?1\ \ldots\ ?k\ \ldots\ ?n,\ \mathcal{P}'$$

Where T_1 and T_2 are CIC terms (types actually) but the lookup considers only the head constant of them. T_2 can also be $FunClass$ and $SortClass$, two pseudo-terms used to represents the type of dependent products and the type of sorts.

The lookup operation gives back the coercion constant c as well as a possibly enriched proof problem $\mathcal{P}'$. The number of metavariables generated to which c is applied to are defined when the coercion is declared. The position of the casted argument is user defined as well, and is returned by the lookup operation. The coerced term has then to be unified with $?k$. Since we allow coercion arguments not to be inferred automatically (like proof obligations) their type may depend on the coerced term (e.g. the proof that the coerced integer is greater than zero has an instance of the coerced integer in its type, and the corresponding metavariable will have index greater then k).

5.2.3 Refinement algorithm

The, essentially first order, unification algorithm presented in [80] is extended with the two following rules that are applied when the second term is the pseudo-term $FunClass$ or $SortClass$.

Unif-with-FunClass

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash \Pi x:?.?\ \overset{\mathcal{R}}{\rightsquigarrow}\ \Pi x:?j.?k:\ s,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash T \overset{?}{\equiv} \Pi x:?j.?k\ \overset{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}'',\ \Sigma''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{?}{\equiv} FunClass\ \overset{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}'',\ \Sigma''}$$

Unif-with-SortClass

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{?}{\equiv} Type(j)\ \overset{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma' \quad (j \text{ fresh})\ \vee \\ \mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{?}{\equiv} Prop\ \overset{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{?}{\equiv} SortClass\ \overset{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'}$$

The rules for the explicit (optional) casts insertions are the following two. In the former one no coercion is inserted, since unification between T_1 and T_2 is successful.

Coerce-to-something-ok

$$\frac{\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{?}{\equiv} T_2 \overset{\mathcal{U}}{\leadsto} \mathcal{P}',\ \Sigma'}{\mathcal{P},\ \Sigma,\ \Gamma \vdash M : T_1 \overset{?}{\equiv} T_2 \overset{\mathcal{C}}{\leadsto} M,\ \mathcal{P}',\ \Sigma'}$$

In the following rule the coercion c is applied to its argument M unifying it with $?k$. The returned term M' can still contains metavariables: $?1 \ldots ?k-1$ may appear in the type of $?k$, thus unifying $?k$ with M may instantiate them (since in the case of dependent types the unification of the types is probably a necessary condition for the unification of the two terms, as claimed by Strecker [89]), but $?k+1 \ldots ?n$ can not appear in the type of $?1 \ldots ?k$, thus they will be probably left in $\mathcal{P}$. Relaxing of the uniform inheritance condition allows to implement subset coercions.

Coerce-to-something-ko

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{?}{\not\equiv} T_2 \\ \mathcal{P},\ \Gamma \vdash T_1 \rightarrowtail T_2 \overset{\Delta}{\leadsto} k,\ c\ ?1\ \ldots\ ?k\ \ldots\ ?n,\ \mathcal{P}' \\ \mathcal{P}',\ \Sigma,\ \Gamma \vdash ?k \overset{?}{\equiv} M \overset{\mathcal{U}}{\leadsto} \mathcal{P}'',\ \Sigma' \\ \mathcal{P}'',\ \Sigma',\ \Gamma \vdash c\ ?1\ \ldots\ ?k\ \ldots\ ?n \overset{\mathcal{R}}{\leadsto} M' : T_2',\ \mathcal{P}''',\ \Sigma'' \\ \mathcal{P}''',\ \Sigma'',\ \Gamma \vdash T_2 \overset{?}{\equiv} T_2' \overset{\mathcal{U}}{\leadsto} \mathcal{P}'''',\ \Sigma''' \end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash M : T_1 \overset{?}{\equiv} T_2 \overset{\mathcal{C}}{\leadsto} M',\ \mathcal{P}'''',\ \Sigma'''}$$

The part of the refinement process that mostly interacts with coercions is the rule for application. The first rule we give is used when the type of the head of an application is completely flexible. This happens frequently during the disambiguation process of Matita, that begins giving a completely flexible type to every constant appearing in the term and progressively instantiates them (see [82]).

We consider n-ary applications, thus a list of arguments $M_1 \ldots M_n$ is processed at once. This choice is also a novelty w.r.t. [80] and is closer to the actual implementation of the refiner in Matita.

Refine-appl-flexible

$$\frac{\begin{array}{ll}\mathcal{P},\ \Sigma,\ \Gamma \vdash N \overset{\mathcal{R}}{\leadsto} N' : ?j,\ \mathcal{P}_1,\ \Sigma_1 & \\ \mathcal{P}_i,\ \Sigma_i,\ \Gamma \vdash M_i \overset{\mathcal{R}}{\leadsto} M'_i :\ T_i,\ \mathcal{P}_{i+1},\ \Sigma_{i+1} & i \in \{1 \ldots n\} \\ \mathcal{P}'' = \mathcal{P}_{n+1} \wedge \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?l :\ \mathit{Type} \wedge \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash ?k :\ ?l; & \\ \mathcal{P}'',\ \Sigma_{n+1},\ \Gamma \vdash ?j \overset{?}{\equiv} \Pi x_1 : T_1. \ldots \Pi x_n : T_n.?k \overset{\mathcal{U}}{\leadsto} \mathcal{P}''',\ \Sigma''' & \end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash N\ M_1\ \ldots\ M_n \overset{\mathcal{R}}{\leadsto} N'\ M'_1\ \ldots\ M'_n : ?k[x_1/M'_1; \ldots; x_n/M'_n],\ \mathcal{P}''',\ \Sigma'''}$$

The line building $\mathcal{P}''$ could be replaced by

$$\mathcal{P}_{n+1},\ \Sigma_{n+1},\ \Gamma \vdash \Pi x_1 : T_1. \ldots \Pi x_n : T_n.? \overset{\mathcal{R}}{\leadsto} Q :\ T,\ \mathcal{P}'',\ \Sigma''$$

and then $?j$ could be unified with Q. We did not adopted that solutions since refining again the whole product type is much more expensive that just generating two fresh metavariables.

The following two rules are used when the head of the application has a type (possibly wrong).

Refine-appl-base

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash N \overset{\mathcal{R}}{\leadsto} N' :\ \Pi x_1 : T_1. \ldots \Pi x_k : T_k.T,\ \mathcal{P}_1,\ \Sigma_1 \\ \left.\begin{array}{l}\mathcal{P}_i,\ \Sigma_i,\ \Gamma \vdash M_i \overset{\mathcal{R}}{\leadsto} M'_i :\ T'_i,\ \mathcal{P}'_i,\ \Sigma'_i \\ \mathcal{P}'_i,\ \Sigma'_i,\ \Gamma \vdash M'_i : T'_i \overset{?}{\equiv} T_i \overset{\mathcal{C}}{\leadsto} M''_i,\ \mathcal{P}_{i+1},\ \Sigma_{i+1}\end{array}\right\} i \in \{1 \ldots n\} \\ \sigma = [x_1/M''_1; \ldots; x_n/M''_n] \\ T' = \Pi x_{n+1} : T_{n+1}\sigma. \ldots .\Pi x_k : T_k\sigma.T\sigma\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash N\ M_1\ \ldots\ M_n \overset{\mathcal{R}}{\leadsto} N'\ M''_1\ \ldots\ M''_n :\ T',\ \mathcal{P}_{n+1},\ \Sigma_{n+1}} (k >= n)$$

Note that the first line hides a possible weak head normalisation phase to check if the type of N' has the shape of a product.

Refine-appl-rec

$$\dfrac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash N \overset{\mathcal{R}}{\leadsto} N' :\ \Pi x_1 : T_1 . \dots \Pi x_k : T_k . T,\ \mathcal{P}_1,\ \Sigma_1 \\ \left.\begin{array}{l}\mathcal{P}_i,\ \Sigma_i,\ \Gamma \vdash M_i \overset{\mathcal{R}}{\leadsto} M_i' :\ T_i',\ \mathcal{P}_i',\ \Sigma_i' \\ \mathcal{P}_i',\ \Sigma_i',\ \Gamma \vdash M_i' : T_i' \overset{?}{\equiv} T_i \overset{\mathcal{C}}{\leadsto} M_i'',\ \mathcal{P}_{i+1},\ \Sigma_{i+1}\end{array}\right\} \quad i \in \{1 \dots k\} \\ T' = T[x_1/M_1''; \dots ; x_k/M_k''] \\ \mathcal{P}_{k+1},\ \Sigma_{k+1},\ \Gamma \vdash N'\ M_1''\ \dots\ M_k'' : T' \overset{?}{\equiv} FunClass \overset{\mathcal{C}}{\leadsto} N'',\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash N''\ M_{k+1}\ \dots\ M_n \overset{\mathcal{R}}{\leadsto} N''' :\ T'',\ \mathcal{P}'',\ \Sigma''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash N\ M_1\ \dots\ M_n \overset{\mathcal{R}}{\leadsto} N''' :\ T''',\ \mathcal{P}'',\ \Sigma''} (k < n)$$

Note that a cast is always attempted, but the rule **Coerce-to-something-ko** inserts a coercion only if needed.

Next two rules insert coercions to adjust the type of abstractions, in case they are not types but terms (and thus their sort is not a sort but a type).

Refine-prod

$$\dfrac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{\mathcal{R}}{\leadsto} T_1' :\ s_1,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash T_1' : s_1 \overset{?}{\equiv} SortClass \overset{\mathcal{C}}{\leadsto} T_1'',\ \mathcal{P}'',\ \Sigma'' \\ \mathcal{P}'',\ \Sigma'',\ \Gamma; x : T_1'' \vdash T_2 \overset{\mathcal{R}}{\leadsto} T_2' :\ s_2,\ \mathcal{P}''',\ \Sigma''' \\ \mathcal{P}''',\ \Sigma''',\ \Gamma; x : T_1'' \vdash T_2' : s_2 \overset{?}{\equiv} SortClass \overset{\mathcal{C}}{\leadsto} T_2'',\ \mathcal{P}'''',\ \Sigma'''' \\ s_3 = \text{Obtained using the PTS rule on the type of } T_1'' \text{ and the type of } T_2''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \Pi x : T_1 . T_2 \overset{\mathcal{R}}{\leadsto} \Pi x : T_1'' . T_2'' :\ s_3,\ \mathcal{P}'''',\ \Sigma''''}$$

Refine-lambda

$$\dfrac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{\mathcal{R}}{\leadsto} T_1' :\ s,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash T_1' : s \overset{?}{\equiv} SortClass \overset{\mathcal{C}}{\leadsto} T_1'',\ \mathcal{P}'',\ \Sigma'' \\ \mathcal{P}'',\ \Sigma'',\ \Gamma; x : T_1'' \vdash M \overset{\mathcal{R}}{\leadsto} M' :\ T,\ \mathcal{P}''',\ \Sigma'''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \lambda x : T_1 . M \overset{\mathcal{R}}{\leadsto} \lambda x : T_1'' . M' :\ \Pi x : T_1'' . T,\ \mathcal{P}''',\ \Sigma'''}$$

The following rules are stated for completeness, but play no role in the process of inserting coercions. The fix point and pattern matching cases will be used, and extended, in Section 5.4.

Refine-var

$$\frac{(x : T) \in \Gamma}{\mathcal{P},\ \Sigma,\ \Gamma \vdash x \overset{\mathcal{R}}{\rightsquigarrow}\ x :\ T,\ \mathcal{P},\ \Sigma}$$

The following rule is expressed in the case of a constant c but is exactly the same if, instead of c, an inductive type constructor k is used.

Refine-constant/constructor

$$\frac{(c : T) \in E}{\mathcal{P},\ \Sigma,\ \Gamma \vdash c \overset{\mathcal{R}}{\rightsquigarrow}\ c :\ T,\ \mathcal{P},\ \Sigma}$$

Inductive type declarations are stored in the environment E together with the number of family parameters (called l) and the number of parameters (called r). The arity of the inductive type is thus $l + r$. The environment lookup operation for inductive types I is thus:

$$l, r, (I : \Pi x_1 : F_1. \dots \Pi x_l : F_l . \Pi x_{l+1} : P_{l+1}. \dots \Pi x_{l+r} : P_{l+1}.s) \in E$$

Refining implicit arguments involves the generation of metavariables. The local substitution attached to each metavariable occurrence is generated here and depends on the context under which the implicit argument is refined. We thus define the following recursive function that generates a list of terms (actually a list of variables, the identity local substitution) taking in input a context.

$$\rho(\Gamma) = \begin{cases} [] & \text{if } \Gamma = [] \\ x :: \rho(\Gamma') & \text{if } \Gamma = x : T; \Gamma' \end{cases}$$

Refine-implicit

$$\frac{\mathcal{P}' = \mathcal{P} \wedge \Gamma \vdash ?l\rho(\Gamma) :\ \mathit{Type} \wedge \Gamma \vdash ?k\rho(\Gamma) : ?l\rho(\Gamma) \wedge \Gamma \vdash ?j\rho(\Gamma) : ?k\rho(\Gamma)}{\mathcal{P},\ \Sigma,\ \Gamma \vdash ? \overset{\mathcal{R}}{\rightsquigarrow} ?j : ?k,\ \mathcal{P}',\ \Sigma}$$

The following rule is used to refine the object of a pattern matching application. It returns not only the refined term but some additional information regarding its inductive type.

Refine-case-step-indtype

$$\frac{\begin{array}{l} l, r, (I : \Pi x_1 : F_1. \ldots \Pi x_l : F_l. \Pi x_{l+1} : P_{l+1}. \ldots \Pi x_{l+r} : P_{l+r}.s) \in E \\ \mathcal{P},\ \Sigma,\ \Gamma \vdash t \overset{\mathcal{R}}{\leadsto} t' :\ T,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash I\ \overbrace{?\ \ldots\ ?}^{l}\ \overbrace{?\ \ldots\ ?}^{r} \overset{\mathcal{R}}{\leadsto} I\ ?u_1\ \ldots\ ?u_{l+r} :\ s,\ \mathcal{P}'',\ \Sigma'' \\ \mathcal{P}'',\ \Sigma'',\ \Gamma \vdash I\ ?u_1\ \ldots\ ?u_{l+r} \overset{?}{\equiv} T \overset{\mathcal{U}}{\leadsto} \mathcal{P}''',\ \Sigma''' \\ \mathcal{P}''',\ \Sigma''',\ \Gamma; x_1 :?v_1; \ldots; x_{i-1} :?v_{i-1} \vdash ?u_i :?v_i \in \mathcal{P}''' \quad i \in \{1 \ldots l + r\} \end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash t \overset{\mathcal{R}_I^E}{\leadsto} t', l, r, u, v, \mathcal{P}''', \Sigma'''}$$

The last line of the previous rule makes explicit that $?v_i$ types $?u_i$. Note that the sort of the inductive type and the sort of $I\ ?u_1\ \ldots\ ?u_{l+r}$ are forced to be the same. In case they are both *Type* they may be distinct universes but having the same constraints with the types F_i and $?v_i$ respectively.

The next rule is used to refine the typing function associated with each pattern matching construct.

Refine-case-step-typef

$$\frac{\begin{array}{l} \mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{\mathcal{R}}{\leadsto} T' :\ A,\ \mathcal{P}',\ \Sigma' \\ B = \Pi x_{l+1} :?v_{l+1}. \ldots . \Pi x_{l+r} :?v_{l+r}. \Pi x : I\ ?u_1\ \ldots\ ?u_l\ x_{l+1}\ \ldots\ x_{l+r}.? \\ \mathcal{P}',\ \Sigma',\ \Gamma \vdash B \overset{\mathcal{R}}{\leadsto} B' :\ Q,\ \mathcal{P}'',\ \Sigma'' \\ \mathcal{P}'',\ \Sigma'',\ \Gamma \vdash A \overset{?}{\equiv} B' \overset{\mathcal{U}}{\leadsto} \mathcal{P}''',\ \Sigma''' \end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{\mathcal{R}_{u,v}^{I,l,r}}{\leadsto} T', \mathcal{P}''', \Sigma'''}$$

The following rule is used to infer the terms R that appear as parameters of the inductive type I if inhabited by the constructor k_i when applies to the family parameters $?u_1 \ldots ?u_l$ and to free metavariable representing the p constructor parameters.

Refine-case-constructor

$$\frac{\begin{array}{l} \mathcal{P},\ \Sigma,\ \Gamma \vdash k_i \overset{\mathcal{R}}{\leadsto} k_i :\ \Pi x_1 : F_1. \ldots . \Pi x_l : F_l. \Pi y_1 : T_1. \ldots . \Pi y_p : T_p. I\ x_1 \ldots x_l\ Q_1\ \ldots Q_r,\ \mathcal{P},\ \Sigma \\ \mathcal{P},\ \Sigma,\ \Gamma \vdash k_i\, ?u_1 \ldots ?u_l\, \overbrace{? \ldots ?}^{p} \overset{\mathcal{R}}{\leadsto} k_i\, ?u_1 \ldots ?u_l\, ?w_1 \ldots ?w_p : I\, ?u_1 \ldots ?u_l\, R_1 \ldots R_r,\ \mathcal{P}',\ \Sigma' \end{array}}{\mathcal{P}, \Sigma, \Gamma \vdash k_i \overset{\mathcal{R}_{u,p}^{I}}{\leadsto} w,\ R,\ \mathcal{P}',\ \Sigma'}$$

The rules **Refine-case-step-indtype**, **Refine-case-step-typef** and **Refine-case-constructor** are used only by the **Refine-case** rule.

Refine-case

$$\dfrac{\begin{array}{l} \mathcal{P},\ \Sigma,\ \Gamma \vdash t \overset{\mathcal{R}_I^E}{\rightsquigarrow} t', l, r, u, v, \mathcal{P}^1, \Sigma^1 \qquad \mathcal{P}^1,\ \Sigma^1,\ \Gamma \vdash T \overset{\mathcal{R}_{u,v}^{I,l,r}}{\rightsquigarrow} T',\ \mathcal{P}_1^2,\ \Sigma_1^2 \\ \left.\begin{array}{l} \mathcal{P}_i^2,\ \Sigma_i^2,\ \Gamma \vdash k_i \overset{\mathcal{R}_{u,p}^I}{\rightsquigarrow} w,\ R,\ \mathcal{P}_i^3,\ \Sigma_i^3 \\ \mathcal{P}_i^3,\ \Sigma_i^3,\ \Gamma \vdash t_i\ ?w_1 \ldots ?w_{p_i} \overset{\mathcal{R}}{\rightsquigarrow} t_i' :\ T_i,\ \mathcal{P}_i^4,\ \Sigma_i^4 \\ \mathcal{P}_i^4,\ \Sigma_i^4,\ \Gamma \vdash T'\, R_1 \ldots R_r\, (k_i\, ?u_1 \ldots ?u_l\, ?w_1 \ldots ?w_{p_i}) \overset{?}{\equiv} T_i \overset{\mathcal{U}}{\rightsquigarrow} \mathcal{P}_{i+1}^2,\ \Sigma_{i+1}^2 \end{array}\right\} \begin{array}{c} i \in \\ \{1 \ldots n\} \end{array} \\ M = \text{match } t' \text{ in } I \text{ return } T'\ [\ k_1\ x_{1_1}\ \ldots\ x_{p_1} \Rightarrow t_1'\ |\ \ldots\ \ |\ k_n\ x_{1_n}\ \ldots\ x_{p_n} \Rightarrow t_n'\ ] \end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \left(\begin{array}{l} \text{match } t \text{ in } I \text{ return } T \\ \quad [k_1\ x_{1_1}\ \ldots\ x_{p_1} \Rightarrow t_1\ |\ \ldots \\ \quad |k_n\ x_{1_n}\ \ldots\ x_{p_n} \Rightarrow t_n] \end{array}\right) \overset{\mathcal{R}}{\rightsquigarrow} M{:}\, T'\ ?u_{l+1} \ldots ?u_{l+r}\ t',\ \mathcal{P}_{n+1}^2,\ \Sigma_{n+1}^2}$$

Recursive definitions are processed with the following rule.

Refine-letrec

$$\dfrac{\begin{array}{l} \mathcal{P}_1 = \mathcal{P} \qquad \Sigma_1 = \Sigma \\ \mathcal{P}_i,\ \Sigma_i,\ \Gamma \vdash \Pi x_{i,1} : T_{i,1}. \ldots \Pi x_{i,p_i} : T_{i,p_i}.T_{i,p_i+1} \overset{\mathcal{R}}{\rightsquigarrow} T_i :\ s_i,\ \mathcal{P}_{i+1},\ \Sigma_{i+1} \quad i \in \{1 \ldots n\} \\ \mathcal{P}_1' = \mathcal{P}_{n+1} \quad \Sigma_1' = \Sigma_{n+1} \\ \left.\begin{array}{l} \mathcal{P}_i',\ \Sigma_i',\ \Gamma; f_1 : T_1; \ldots ; f_n : T_n \vdash t_i \overset{\mathcal{R}}{\rightsquigarrow} t_i' :\ T_i',\ \mathcal{P}_i'',\ \Sigma_i'' \\ \mathcal{P}_i'',\ \Sigma_i'',\ \Gamma \vdash T_i \overset{?}{\equiv} T_i' \overset{\mathcal{U}}{\rightsquigarrow} \mathcal{P}_{i+1}',\ \Sigma_{i+1}' \end{array}\right\} \quad i \in \{1 \ldots n\} \end{array}}{\begin{array}{l} \mathcal{P}_1,\ \Sigma_1,\ \Gamma \vdash \left(\begin{array}{l} \text{letrec } f_1(x_{1,1}{:}T_{1,1}) \ldots (x_{1,p_1}{:}T_{1,p_1}){:}T_{1,p_1+1} \text{ on } l_1 := t_1 \text{ and } \ldots \\ \text{and } f_n(x_{n,1}{:}T_{n,1}) \ldots (x_{n,p_n}{:}T_{n,p_n}){:}T_n \text{ on } l_n := t_n \text{ in } f_j \end{array}\right) \overset{\mathcal{R}}{\rightsquigarrow} \\ \left(\begin{array}{l} \text{letrec } f_1(x_{1,1}{:}T_{1,1}') \ldots (x_{1,p_1}{:}T_{1,p_1}'){:}T_{1,p_1+1}' \text{ on } l_1 := t_1' \text{ and } \ldots \\ \text{and } f_n(x_{n,1}{:}T_{n,1}') \ldots (x_{n,p_n}{:}T_{n,p_n}'){:}T_n' \text{ on } l_n := t_n' \text{ in } f_j \end{array}\right) :\ T_j',\ \mathcal{P}_{i+1}',\ \Sigma_{i+1}' \end{array}}$$

Note that the types of the functions f_i live in context Γ while their bodies live in $\Gamma; f_1 : T_1; \ldots ; f_n : T_n$. Although the unification step

$$\mathcal{P}_i'',\ \Sigma_i'',\ \Gamma \vdash T_i \overset{?}{\equiv} T_i' \overset{\mathcal{U}}{\rightsquigarrow} \mathcal{P}_{i+1}',\ \Sigma_{i+1}'$$

is performed in context Γ, thus the types of the bodies T_i' can not depend on the functions f_i.

5.2.4 Partial transitive closure

The transitive closure of the coercion graph may lead to an infinite graph. For this reason many implementations drop that idea, and compute long paths of coercions on the fly. This approach, used by Saibi [84] and Bailey [7] for COQ and Lego, has been successfully used in the formalisation of the fundamental theorem of algebra [34]. From the user point of view, since coercions are usually hidden, having long chain of coercions or just one is exactly the same. Looking at how the typechecker/refiner of MATITA works: having just one coercion is much faster. The typechecking algorithm of MATITA reconstructs the environment (containing all constants appearing the typechecked term) on the fly [24, 80]. Objects are loaded when required and some sort of trusting mechanism is usually employed. Objects previously typechecked and stored on disk are not typed again when loaded, they declare their type that has been previously certified. This allows, unless a δ-reduction step is involved in a conversion test, not to load the constants used by an object in its proof (or body, if it is a definition).

Our approach, similar to the one used by Callaghan [20], is to compute a partial transitive closure of the coercions graph. This algorithm can not be complete, but composite coercions not generated automatically by the system can be manually added, and an ad-hoc tactic called compose eases this task. The judgement

$$\mathcal{P},\ \Gamma \vdash T_1 \rightarrowtail T_2 \overset{\Delta}{\leadsto} k,\ c\ ?1\ \ldots\ ?k\ \ldots\ ?n,\ \mathcal{P}'$$

used in the extended unification rules does not recursively look for a path longer than one step from T_1 to T_2. Ideally, if the coercion graph is transitively closed, this simple search is sufficient. This search is in general faster than the recursive one, since all paths are precomputed.

This approach, in combination with the choice of allowing multiple paths between the same types, clearly leads to new problems that arise when typechecking is performed (since terms are fully specified, with no missing information like in [11]). For example, consider the coherent coercion graph of Figure 5.1, where $\Phi = \phi_1 \circ \phi_2$ and $\Psi = \psi_1 \circ \psi_2$.

$$\begin{tikzcd}
 & \boxed{\mathit{Top}} \arrow[dl, "\phi_2"'] \arrow[dr, "\psi_2"] \arrow[dd, bend right, "\Phi"] \arrow[dd, bend left, "\Psi"'] & \\
\boxed{\mathit{Left}} \arrow[dr, "\phi_1"'] & & \boxed{\mathit{Right}} \arrow[dl, "\psi_1"] \\
 & \boxed{\mathit{Bottom}} &
\end{tikzcd}$$

Figure 5.1: Simple (coherent) coercion graph

Unification may face the following problem:

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash \Phi\ ?j \overset{?}{\equiv} \phi_1\ ?k \overset{\mathcal{U}}{\leadsto} \mathcal{P}',\ \Sigma'$$

Since both Φ and ϕ_1 are rigid term in weak head normal form unification can fail, not performing a possibly expensive δ-expansion. The unification algorithm usually knows nothing about coercions, thus unfolding Φ can not be considered an hint. Another example is

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash \phi_1\ ?j \overset{?}{\equiv} \psi_1\ ?k \overset{\mathcal{U}}{\leadsto} \mathcal{P}',\ \Sigma'$$

Both ϕ_1 and ψ_1 are rigid terms already in weak head normal form. Again unification fails, since it is not aware of the coherence property of the coercion graph. We give a better explanation of these problems in Section 5.3.4 and we propose an extension to the unification algorithm that has been implemented and effectively used in MATITA to solve these problems.

Generation of composite coercions

Following Callaghan's proposal, every time a coercion c is declared by the user we extend the coercion graph Δ adding the following, informally defined, set:

$$\{c_i \circ c \circ c_j | c_i \in \Delta \wedge c_j \in \Delta\} \cup \{c \circ c_j | c_j \in \Delta\} \cup \{c_i \circ c | c_i \in \Delta\}$$

Clearly not all c_i and c_j in Δ are good candidates for a composition with c. A coarse filtering can be done looking at the source and target types (actually their label approximation). We try to generate all composite coercions that match that criteria, except the ones that have as source and target the same non parametric type. For example, when a coercion from natural numbers to integers is declared together with a (subset) coercion from integers to natural numbers we never generate a composite coercion from and to natural numbers. In case this coercion makes sense the user can declare it by hand, but in our experience this filtering policy drops only coercions that make little sense. The description of this process for generating composite coercions is a novelty to the author's knowledge.

Let coercions c and d be defined as follows:

$$\emptyset, [] \vdash S\ x_{s_1}\ \dots\ x_{s_m} \rightarrowtail Q\ x_{Q_1}\ \dots\ x_{Q_n} \overset{\Delta}{\rightsquigarrow} k_c, c\ ?_{c_1}\ \dots\ ?_{c_o},\ \mathcal{P}$$
$$\emptyset, [] \vdash Q\ y_{Q_1}\ \dots\ y_{Q_n} \rightarrowtail T\ x_{T_1}\ \dots\ x_{T_p} \overset{\Delta}{\rightsquigarrow} k_d, d\ ?_{d_1}\ \dots\ ?_{d_q},\ \mathcal{P}'$$

To obtain the composite coercions $c \circ d$ the following refinement is performed.

$$\emptyset, [], [] \vdash \overbrace{\lambda x_1 : ?. \dots \lambda x_v : ?.}^{k_d+o} d\ \overbrace{?\ \dots\ ?}^{k_d}\ (c\ \overbrace{?\ \dots\ ?}^{o}) \overset{\mathcal{R}}{\rightsquigarrow}$$
$$\lambda x_1 : ?1. \dots \lambda x_v : ?v.d\ ?(v+1)\ \dots\ ?w\ (c\ ?(w+1)\ \dots\ ?(w+o)),\ \mathcal{P},\ \Sigma$$

Note that $v = k_d + o$, $w = 2 * k_d + o$, $\mathcal{P}$ contains more than $w + o$ metavariables since it also contains their type and Σ may be not empty. For example, consider the two following simple coercions:

```
definition L2V :  ∀l : List.Vector (length l)
definition V2N :  ∀n : nat.Vector n → nat
```

We build nat_OF_List := L2V∘V2N since the label for the target type of L2V is Vector, the same for the input type of V2N. The composed coercions, before the refinement operation, is

```
definition nat_OF_List := λl:?.λ n:?. V2N ? (L2V ?)
```

Some heuristic is used to guess the names in the *lambda* spine such that they are possibly similar to the ones used in the two coercions types.

This term is refined obtaining the following term (where the substitution Σ has been applied).

```
definition nat_OF_List :=
  λ l :?4[]. λ n:?7[l]. V2N (len ?13[l ; n ; A]) (L2V ?13[l ; n ; A])
```

Note that Σ instantiated the first argument of V2N with (length ?13). The resulting proof problem $\mathcal{P}$, substituted with Σ follows. Note that metavariables appearing as types of the abstracted variables have no constraints at all and the refiner just created a pile of metavariables to type them. At the top of that pile there is a fresh universe Type.

```
⊢ ?4: ?3[]
⊢ ?3: ?2[]
⊢ ?2: Type
l : ?4[] ⊢ ?7: ?6[l]
l : ?4[] ⊢ ?6: ?5[]
⊢ ?5: Type
⊢ ?13: List
```

We now build a substitution for the metavariables that appear in the body of the composed coercion and their types to the initial segment of the λ spine. We use the order relation $\ll_{\mathcal{P}}$ to topologically sort them and decide which has to be bound to the first abstraction and so on.

$$\text{skip\_lambdas}(n,\ t) = \begin{cases} \text{skip\_lambdas}(n+1,\ b) & \text{if } t = \lambda x : T.b \\ n,\ t & \text{otherwise} \end{cases}$$

In MATITA variables are represented with De Bruijn indexes, thus the function $\text{rel}(j)$ is the name of the variable bound at distance j.

$$\text{mksubs}(j,\ \mathcal{P}) = \begin{cases} [] & \text{if } \mathcal{P} \text{ is empty} \\ ?k := \text{rel}(j) :: \text{mksubs}(j-1,\ \mathcal{P}') & \text{if } \mathcal{P} = \Gamma \vdash ?k : T_{?k} :: \mathcal{P} \end{cases}$$

Thus the complete algorithm to create the substitution for the term t living in $\mathcal{P}$ and Σ is the following pseudo-code (where filter is the function that filters a proof problem according to a predicate and $\mathcal{M}$ is the function that collects the metavariables occurring in a term given in Definition 5.2).

$$\begin{array}{l}\text{let } n,\ t = \text{skip\_lambdas}(\Sigma(t)) \text{ in}\\ \text{let } \mathcal{P}' = \text{filter}(\mathcal{P}, \lambda(\Gamma \vdash ?j : T).j \in \mathcal{M}(t)) \text{ in}\\ \text{mksubst}(n,\ \text{topological\_sort}(\mathcal{P}',\ \ll_{\mathcal{P}}))\end{array}$$

In our example only ?13 appears in the body of the composed coercion and its type is closed, thus ?13 will be bound to the first abstraction l.

We then refine again the term and we prune unused lambdas obtaining the following term:

```
nat_OF_List : List → nat := λ l:List.V2N (length l) (L2V l)
```

We now need to calculate the k value for nat_OF_Vector. It represent the abstraction that bounds the input of the newly generated coercion. We have to look at the parameters of the inner coercion and find to which lambda the parameter occupying the k position (of that coercion) is bound. In our example k is trivially one.

This procedure is not complete since the refinement process is not complete itself depending on higher order unification. Composing total coercions works pretty well according to our experience, what still needs to be tuned is the composition of subset coercions. An heuristic to avoid polluting the coercion graph is to never automatically compose two subset coercions. This implementation of coercions is currently used in the ongoing formalisation of the Lebesgue's dominated convergence theorem, a part of the D.A.M.A.[2] project.

5.3 Implementing mathematical structures

It is well known that formalising mathematical concepts in type theory is not straightforward, and one of the most used metrics to describe this difficulty is the gap (in lines of text) between the pen&paper proof, and the formalised version. A

[2]Dimostrazione Assistita per la Matematica e l'Apprendimento (D.A.M.A.).
http://dama.cs.unibo.it/

motivation for that may be that many intuitive concepts widely used in mathematics, like graphs for example, have no simple and handy representation (see for example the complex hypermap construction used to describe planar maps in the four colour theorem [44]). On the contrary, some widely studied fields of mathematics do have a precise and formal description of the objects they study. The most well known one is algebra, where a rigorous hierarchy of structures is defined and investigated. One may expect that formalising algebra in an interactive theorem prover should be smooth, and that the so called De Bruijn factor should be not so high for that particular subject.

Many papers in the literature [34] give evidence that this is not the case. In this paper we analyse some of the problems that arise in formalising a hierarchy of algebraic structures and we propose a general mechanism that allows to tighten the distance between the algebraic hierarchy as is conceived by mathematicians and the one that can be effectively implemented in type theory.

In particular, we want to be able to formalise the following informal piece of mathematics[3] *without making more information explicit*, expecting the interactive theorem prover to understand it as a mathematician would do.

Example 5.1 Let k be an ordered field. An ordered vector space over k is a vector space V that is also a poset at the same time, such that the following conditions are satisfied

1. for any $u, v, w \in V$, if $u \leq v$ then $u + w \leq v + w$,
2. if $0 \leq u \in V$ and any $0 < \lambda \in k$, then $0 \leq \lambda u$.

Here is a property that can be immediately verified: $u \leq v$ iff $\lambda u \leq \lambda v$ for any $0 < \lambda$. □

We choose this running example instead of the most common example about rings[34, 74, 11] because we believe the latter to be a little deceiving. Indeed, a ring is usually defined as a triple (C,+,∗) such that (C,+) is a group, (C,∗) is a

[3]PlanetMath, definition of Ordered Vector Space.

semigroup, and some distributive properties hold. This definition is imprecise or at least not complete, since it does not list the neutral element and the inverse function of the group. Its real meaning is just that a ring is an *additive* group that is also a *multiplicative* semigroup (on the same carrier) with some distributive properties. Indeed, the latter way of defining structures is often adopted also by mathematicians when the structures become more complex and embed more operations (e.g. vector spaces, Riesz spaces, integration algebras).

Considering again our running example, we want to formalise it using the following syntax, and we expect the proof assistant to interpret it as expected:

```
record OrderedVectorSpace : Type := {
 V:> VectorSpace; (* we suppose that V.k is the ordered field *)
 p:> Poset with p.CA_po = V.CA_vs;
 add_le_compat: ∀u,v,w:V. u ≤ v → u + w ≤ v + w;
 mul_le_compat: ∀u:V.∀α :k. 0 ≤ u → 0 < α → 0 ≤ α ∗ u
}.
lemma trivial: ∀R.∀u,v:R. (∀α . 0 < α→ α ∗ u ≤ α ∗ v) → u ≤ v.
```

The first statement declares a *record type*. A record type is a labelled telescope. A *telescope* is just a generalised Σ-type. Inhabitants of a telescope of length n are heavily typed n-tuples $\langle x_1, \ldots, x_n\rangle_{T_1,\ldots,T_n}$ where x_i must have type $T_i x_1 \ldots x_{i-1}$. The heavy types are necessary for type reconstruction. Inhabitants of a record type with n fields are just labelled n-tuples $\langle x_1, \ldots, x_n\rangle_R$ where R is a reference to the record type declaration, which declares once and for all the types of fields. Thus terms containing inhabitants of records are smaller and require less type-checking time than their equivalents that use telescopes.

Beware of the differences between our records — which are implemented, at least as telescopes, in most systems — and *dependently typed records* “à la Betarte/Tasistro/Pollack” [16, 15, 31]:

1. there is no “dot” constructor to uniformly access by name fields of any record; however, we suppose that ad-hoc projections are automatically declared by the system;

2. there is no structural subtyping relation “à la Betarte/Tasistro” between records; however, ad-hoc coercions “à la Pollack” can be declared by the user; in particular, we suppose that when a field is declared using “:>”, the relative projection is automatically declared as a coercion by the system;

3. there are no manifest fields “à la Pollack”; the **with** notation is usually understood as syntactic sugar for declaring on-the-fly a new record with a manifest field; however, having no manifest fields in our logic, we will need a different explanation for the **with** type constructor.

When lambda-abstractions and dependent products do not type their variable, the type of the variable must be inferred by the system during type reconstruction. Similarly, all mathematical notation (e.g. “$*$”) hides the application of one projection to a record (e.g. “?.$*$” where ? is a placeholder for a particular record). The notation “x:R” can also hide a projection R.CA from R to its carrier.

All projections are monomorphic, in the sense that different structures have different projections to their carrier. All placeholders in projections must be inferred during type reconstruction. This is not a trivial task: in the expression “$\alpha * \mathrm{u} \leq \alpha * \mathrm{w}$” both sides of the inequation are applications of the scalar product of some vector space R (since u and v have been previously assigned the type R.CA); since their result are compared, the system must deduce that the vector space R must also be a poset, hence an ordered vector space.

In the rest of this section we address the problem of representing mathematical structures in a proof assistant which: 1) is based on a type theory with dependent types, telescopes and a computational version of Leibniz equality; 2) implements coercive subtyping, accepting multiple coherent paths between type families; 3) implements a restricted form of higher order unification and type reconstruction. Lego, Coq, Plastic and MATITA are all examples of proof assistants based on such theories. In the next sections we highlight one by one the problems that all these systems face in understanding the syntax of the previous example, proposing solutions that require minimal modifications to the implementation.

5.3.1 Telescopes and computational Leibniz equality

In the following we implement our construction using telescopes and computational Leibniz equality.

Telescopes can be implemented over non recursive CIC inductive types with a single constructor, and syntactic sugar for them is supported by MATITA.

Computational Leibniz equality is such that a rewriting step may compute under certain conditions. Rewriting steps in CIC are usually applications of that term:

```
eq_rect : ∀A:Type.∀x:A. ∀P:A →Type.(P x) →∀y:A. x=y → (P y) :=
  λA:Type.λx:A.λP:A →Type.λf:P x.λy:A.λe:x=y. match e with [ refl ⇒f ]
```

When e is a closed term, since = can only be inhabited by refl, the rewriting step computes to the rewritten term (f here). It also works for open proofs over types with a decidable equality. We will benefit from this computational behaviour because of the conversion rule that can identify a term and its rewritten if the equality proof is closed.

5.3.2 Weakly manifest telescopes

We drop Σ/Ψ types in favour of primitive records, whose inhabitants do not require heavy type annotations. However, we are back at the problem of manifest fields: every time the user declares a record type with n fields, to follow closely the approach of Pollack the system should declare 2^n record types having all possible combinations of manifest/opaque fields.

To obtain a new solution for manifest fields we exploit the fact that manifest fields can be declared using **with** and we also go back to the intuition that records with and without manifest fields should all have the same representation. That is, when x ≡3 (x is definitionally equal to 3) and p: P x, the term $\langle x,\ p\rangle_R$ should be both an inhabitant of the record R := { n: nat; H: P n} and of the record R **with** n = 3. Intuitively, the **with** notation should only add in the context the new "hypothesis" x ≡3. However, we want to be able to obtain this effect without extending the type theory with **with** and without adding at run time new equations to the convertibility check. This is partially achieved by approximating x ≡3 with an hypothesis of type x = 3 where "=" is Leibniz polymorphic equality.

To summarise, the idea is to represent an inhabitant of R := {n: nat; H: P n} as a pair $\langle x,\ p\rangle_{R}$ and an inhabitant of R **with** n=3 as a couple $\langle c,\ q\rangle_{R,\lambda c:R.c.n=3}$ of type Σ c:R. c.n=3. Declaring the first projection of the Σ-type as a coercion, the system is able to map every element of R **with** n=3 into an element of R.

However, the illusion is not complete yet: if c is an inhabitant of R **with** n=3, c.1.n (that can be written as c.n because .1 is a coercion) is Leibniz-equal to 3 (because of c.2), but is not convertible to 3. This is problematic since terms that were well-typed in the system presented by Pollack are here rejected. Several concrete example can already be found in our running example: to type u + w ≤ v + w (in the declaration of add_le_compat), the carriers p.CA_{po} and V.CA_{vs} must be convertible, whereas they are only Leibniz equal. In principle, it would be possible to avoid the problem by replacing u + w ≤ v + w with $[u+w]_{p.2}$≤$[v+w]_{p.2}$ where $[\_]_{\_}$ is the constant corresponding to Leibniz elimination, i.e. $[x]_{w}$ has type Q[M] whenever x has type Q[N] and w has type N=M. However, the insertion of these constants, even if done automatically with a couple of mutual coercions, makes the terms much larger and more difficult to reason about.

5.3.3 Manifesting coercions

To overcome the problem, consider c of type R **with** n=3 and notice that the lack of conversion can be observed only in c.1.n (which is not definitionally equal to 3) and in all fields of c.1 that come after n (for instance, the second field has type P c.1.n in place of P 3). Moreover, the user never needs to write c.1 anywhere, since c.1 is declared as a coercion. Thus we can try to solve the problem by declaring a different coercion such that c.1.n is definitionally equal to 3. In our example, the coercion[4] is

definition k_R^n : ∀m : nat. R **with** n=m → R :=

[4]The name of the coercion is k_R^n verbatim, R and n are not indexes.

λ m:nat. λ x:Σc:R.c.n=m. $\langle$m, $[x.1.H]_{x.2}\rangle_R$

Once k^n_R is declared as a coercion, c.H is interpreted as (k^n_R 3 c).H which has type P (k^n_R 3 c).n, and that is now definitionally equal to P 3. Note also that (k^n_R 3 c).H is definitionally equal to $[c.1.H]_{c.2}$ that is definitionally equal to c.1.H when c.2 is a closed term of type c.1.n = 3. When the latter holds, c.1.n is also definitionally equal to 3, and the manifest type information is actually redundant, according to the initial intuition. The converse holds when the system is proof irrelevant, or, with minor modifications, when Leibniz equality is stated on a decidable type [53].

Coming back to our running example regarding ordered vector spaces, u + w $\leq$ v + w can now be parsed as the well-typed term

$$\mathrm{u\ (V.+)\ w\ ((k^{CA_{po}}_{Poset}\ V.CA_{vs}\ p).\leq)\ v\ (V.+)\ w}$$

Things get a little more complex when **with** is used to change the value of a field f1 that occurs in the type of a second field f2 that occurs in the type of a third field f3. Consider the record type declaration R := { f_1: T; f_2: P f_1; f_3: Q f_1 f_2} and the expression R **with** f_1= M, interpreted as Σc:R. c.f_1= M. We must find a coercion from R **with** f_1= M to R declared as follows

definition $k^{f_1}_R$: ∀M:T. R **with** f_1 = M → R :=
  λM:T. λx:Σc:R.c.f1=M. $\langle$M, $[c.1.f_2]_{c.2}$, w$\rangle$

for some w that inhabit Q M $[c.1.f_2]_{c.2}$ and that must behave as c.1. f_3 when c.1. f_1≡M. Observe that c.1. f_3 has type Q c.1.f_1 c.1.f_2, which is definitionally equivalent to Q c.1.f_1 $[c.1.f_2]_{refl_T\ c.1.f_1}$, where $refl_T$ c.1.f_1 is the canonical proof of c.1. f_1= c.1.f_1. Thus, the term w we are looking for is simply $[[c.1.f_3]]_{c.2}$ which has type Q M $[c.1.f_2]_{c.2}$ where $[[\_]]_\_$ is the constant corresponding to *computational dependent elimination* for Leibniz equality:

lemma $[[\_]]_p$: Q x ($refl_A$ x) →Q y p.
where x : A, y : A, p : x = y, Q : (∀z. x = z → Type) and $[[M]]_{refl_A\ x}$ ≡ M.

To avoid handling the first field differently from the following, we can always use $[[\_]]_\_$ in place of $[\_]_\_$.

The following derived typing and reduction rules show that our encoding of **with** behaves as expected.

Phi-Start

$$\frac{}{\vdash \emptyset \text{ valid}}$$

Phi-Cons

$$\frac{\vdash \Phi \text{ valid} \quad R, l_1, \ldots, l_n \text{ free in } \Phi \qquad T_i : \Pi l_1 : T_1. \ldots .\Pi l_{i-1}.T_{i-1}.Type \quad i \in \{1, \ldots, n\}}{\vdash \Phi, R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type \text{ valid}}$$

Form

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi \quad \Gamma, l_1 : T_1, \ldots, l_{i-1} : T_{i-1} \vdash a : T_i}{\Gamma \vdash R \text{ with } l_i = a : Type}$$

Intro

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi \quad \Gamma \vdash R \text{ with } l_i = a : Type \qquad \Gamma \vdash M_k : T_k\ M_1\ \ldots\ M_{i-1}\ a\ M_{i+1}\ \ldots\ M_{k-1} \quad k \in \{1, \ldots, i-1, i+1, \ldots, n\}}{\Gamma \vdash \langle\langle M_1, \ldots, M_{i-1}, a, M_{i+1}, \ldots, M_n\rangle_R, refl_A\ a\rangle_{R,\lambda r:R.a=a} : R \text{ with } l_i = a}$$

Coerc

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi \qquad \Gamma \vdash R \text{ with } l_i = a : Type \quad \Gamma \vdash c : R \text{ with } l_i = a}{\Gamma \vdash k_R^{l_i}\ a\ c : R}$$

Coerc-Red

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi}{\Gamma \vdash k_R^{l_i}\ a\ \langle\langle M_1, \ldots, M_n\rangle_R, w\rangle_{R,s} \rhd \langle M_1, \ldots, M_{i-1}, a, [[M_{i+1}]]_w, \ldots, [[M_n]]_w\rangle_R}$$

Proj

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi \qquad \Gamma \vdash R \text{ with } l_i = a : Type \quad \Gamma \vdash c : R \text{ with } l_i = a}{\Gamma \vdash (k_R^{l_i}\ a\ c).l_j : T_j\ (k_R\ a\ c).l_1\ \ldots\ (k_R\ a\ c).l_{j-1}}$$

Proj-Red1

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ \langle\langle M_1, \ldots, M_n\rangle_R, w\rangle_{R,s}).l_j \rhd M_j}\ j < i$$

Proj-Red2

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ c).l_i \rhd a}$$

Proj-Red3

$$\frac{\vdash \Phi \text{ valid} \quad (R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ \langle\langle M_1, \ldots, M_n\rangle_R, w\rangle_{R,s}).l_j \rhd [[M_j]]_w}\ i < j$$

5.3.4 Carrier sharing and unification

Following the ideas from the previous section, we can implement **with** as syntactic sugar: R **with** f = M is parsed as Σ c:R.c.f = M and our special coercion $\mathrm{k}_\mathrm{R}^\mathrm{f}$ respecting definitional equality is defined from Σ c:R. c.f=M to R. The scope of the special coercion should be local to the scope of the declaration of a variable of type R **with** f=M. When **with** is used in the declaration of one record field, as in our running example, the scope of the coercion extends to the following record fields, and also to the rest of the script.

As our running example shows, one important use of **with** is in multiple inheritance, in order to share multiply imported sub-structures. For instance, a ordered vector space inherits from a partially ordered set and from a vector space, under the hypothesis that the two carriers (singleton structures) are shared. Since sub-typing is implemented by means of coercions, multiple inheritance with sharing induces multiple coercion paths between nodes in the coercion graph (see Figure 6.1, dotted lines hide intermediate structures like groups and monoids). When the system needs to insert a coercion to map an inhabitant of one type to an inhabitant of a super-type, it must choose one path in the graph. In order to avoid random choices that lead to unwanted interpretations and to type errors (in systems having dependent types), *coherence* of the coercion graph is usually required [58, 11]. The graph is coherent when the diagram commutes according to $\beta\eta$-conversion. However in the following we drop η-conversion which is not supported in MATITA.

$$\begin{tikzcd}
 & & f-algebra \arrow[dl, "r"'] \arrow[dr, dashed] & & \\
 & OrderedVectorSpace \arrow[dl, "V"'] \arrow[dr, "p"] & & Algebra \arrow[dl, dashed] \arrow[dr, dashed] & \\
Poset \arrow[dr, "CA_{po}"'] & & VectorSpace \arrow[dl, "CA_{vs}"'] & & Ring \arrow[dlll, dashed, bend left] \\
 & Type & & &
\end{tikzcd}$$

Figure 5.2: Inheritance graph from the library of MATITA

One interesting case of multiple coherent paths in a graph is constituted by coherent paths between two nodes and the arcs between them obtained by composition of the functions forming one path. Indeed, it is not unusual in large formalisation as CoRN [34] to have very deep inheritance graphs and to need to cast inhabitants of very deep super-types to the root type. For instance, the expression ∀x: R should be understood as ∀x: k R where k is a coercion from the ordered, archimedean, complete field of real numbers to its carrier. Without composite coercions, the system needs to introduce a coercion to ordered, archimedean fields, then another one to ordered fields, another one to fields, and then to rings, and so on, generating a very large term and slowing down the type-checker.

If coherent DAGs of coercions pose no problem to conversion, they do for unification, although this aspect has been neglected in the literature. In particular, consider again our running example, whose coercion graph is shown in Fig. 6.1. Suppose that the user writes the following (false) statement: ∀x.-x $\leq$x where -x is just syntactic sugar for -1 $*$ x. The statement will be parsed as ∀x:$?_1$.-1 $?_4$.$*$ x $?_5$.$\leq$ x and the type reconstruction engine will produce the following two unification constraints: $?_1 \overset{?}{\equiv} ?_4$.CA$_{vs}$ (since x is passed to $?_4$.$*$) and $?_4$.CA$_{vs} \overset{?}{\equiv} ?_5$.CA$_{po}$ (since -1 $*$ x is passed to $?_5$.$\leq$). The first constraint is easily solved, "discovering" that x should be an element of a vector space, or a element of one super-structure of a vector space (since $?_4$ can still be instantiated with a coercion applied to an element of a super-structure). However, the second constraint is problematic since it asks to unify two applications ($?_4.\mathrm{CA}_{vs}$ and $?_5.\mathrm{CA}_{po}$) having different heads. When full higher-order unification is employed, the two heads (two projections) are unfolded and unification will eventually find the right unifier. However, unfolding of constants during unification is too expensive in real world implementations, and higher order unification is never implemented in full generality, preferring an incomplete, but deterministic unification strategy.

Since expansion of constants is not performed during unification, the constraint to be solved is actually a rigid-rigid pattern with two different heads. To avoid failure, we must exploit the coherence of the coercion graph. Indeed, since the arguments of the coercions are metavariables, they can still be instantiated with any possible path in the graph (applied to a final metavariable representing the structure the path is applied to). For instance, $?_4.\mathrm{CA}_{vs}$ can be instantiated to $?_6.\mathrm{p}.\mathrm{CA}_{vs}$ where $?_6$ is a ordered vector space and the vector space $?_4$ is obtained from $?_6$ forgetting the poset structure.

Thus the unification problem is reduced to finding two coherent paths in the graph ending with CA_{vs} and CA_{po}. A solutions is given by paths $?_6.\mathrm{V}.\mathrm{CA}_{po}$ and $?_6.\mathrm{p}.\mathrm{CA}_{vs}$. Another one by $?_7.\mathrm{r}.\mathrm{V}.\mathrm{CA}_{po}$ and $?_7.\mathrm{r}.\mathrm{p}.\mathrm{CA}_{vs}$ where $?_7$ is an f-algebra.

Among all solutions the most general one corresponds to the *pullback* in categorical terms) of the two coercions, when it exists. In the example, the pullback is given by V and p. All other solutions (e.g. r.V and r.p) factor through it.

If the pullback does not exist (i.e. there are different solutions admitting anti-unifiers), the system can just pick one solution randomly, warning the user about the arbitrary choice. Coercion graphs for algebraic structures usually enjoy the property that there exists a pullback for every pairs of coercions with the same target.

Finally, note that the Coq system does not handle composite coercions, since these would lead to multiple paths between the same types. However, long chains of coercions are somehow problematic for proof extraction. According to private communication, an early experiment in auto-packing chains of coercions was attempted, but dropped because of the kind of unification problems just explained. After implementing the described procedure for unification up to coherence in Matita, we were able to implement coercion packing.

5.3.5 Type reconstruction

Syntactic de-sugaring of **with** expression for a large hierarchy of mathematical structures has been made by hand in MATITA, proving the feasibility of the approach. In particular, combining de-sugaring with the unification up to coherence procedure described in the previous paragraph, we are able to write the first part of our running example in MATITA.

The statement of the trivial lemma, however, is not accepted yet. The type reconstruction algorithm has been described in Section 5.2.3. Insertion of coercions interacts badly with open types. Consider, for instance, the following example and assume a coercion from natural numbers to integers.

```
∀P : int → Prop. ∀Q : nat → Prop. ∀b. P b ∧Q b.
```

Here P b is processed before Q b. The rules **Refine-appl-base** and **Refine-appl-rec** never apply a cast around arguments with a flexible types, since the **Coerce-to-something-ok** rule always apply. Then Q b is processed, but now b has type int, int $\overset{?}{\not\equiv}$ nat and there is no coercion from int to nat. The problem here is that a coercion was needed around the first occurrence of b but since its type was flexible **Coerce-to-something-ko** was not triggered.

To solve the problem, one important step is the realization that rules that insert coercions and rules that do not are occurrences of the same rule when identities are considered as coercions. In [22, 23], Chen proposes an unified set of rules that also employes least upper bounds ($\mathcal{LUB}$) in the coercion graph to compute less coerced solutions. Chen's rule for application adapted with metavariables in place of coercions is the following:

Chen-appl-always-coerce

$$\frac{\begin{array}{c}\mathcal{P},\ \Sigma,\ \Gamma \vdash f \overset{\mathcal{R}}{\leadsto} f' : C,\ \mathcal{P}',\ \Sigma' \quad \mathcal{P}'' = \mathcal{P}',\ \Sigma,\ \Gamma \vdash ?^c_i : C \overset{\mathcal{LUB}}{\to} \Pi x : A.B \\ \mathcal{P}'',\ \Sigma',\ \Gamma \vdash a \overset{\mathcal{R}}{\leadsto} a' : A',\ \mathcal{P}''',\ \Sigma'' \quad \mathcal{P}'''' = \mathcal{P}''', \Gamma \vdash ?^c_j : A' \to A\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash f\ a \overset{\mathcal{R}}{\leadsto} (?^c_i\ f')\ (?^c_j\ a') : B[(?^c_j\ a')/x]\mathcal{P}'''',\ \Sigma''}$$

Metavariables marked with $\mathbf{.}^c$ can only be instantiated with coercions. Adopting this rule, the problematic example above is accepted since it is understood as:

```
∀P : int → Prop. ∀Q : nat → Prop. ∀b: ?1. (?c2 P) (?c3 b) ∧(?c4 Q) (?c5 b)
```

where ?1 can be instantiated with nat, $?^c3$ with the coercion from nat to int and all other $?^c$ metavariables with the identity.

From this example it is clear that Chen's rule modified with metavariables is able to type every term generating a large number of constraints that must inefficiently be solved at the very end looking at the coercion graph.

Note, however, that rule **Chen-appl-always-coerce** in its full generality is not required to accept our running example. We believe this not to be a coincidence. Indeed, most formulae in the algebraic domain are of a particular shape:

1. universal quantifications are either on structure types (e.g. $\forall G$: Group) or elements of some structure (e.g. $\forall g : G$ to be understood as $\forall g : G.CA$);

2. functions either take structures in input (e.g. $G \times G$); or they manipulate structure elements whose domain is left implicit (e.g. exponentiation $\_^{\_} : M.CA \to nat \to M.CA$ for some monoid M). In particular, all operations in a structure are functions of the second kind.

Under this assumption, rule **Chen-appl-always-coerce** can be relaxed to our **Refine-appl-***, in which $\overset{\mathcal{C}}{\leadsto}$ is replaced with $\overset{\mathcal{C}_{\mathcal{LUB}}}{\leadsto}$. Moreover new rules for explicitly and implicitly typed universal quantification are added. We will mark metavariables with $\mathbf{.}^s$ to state that they can be instantiated only with sorts.

The rule **Refine-lambda** is replaced by the following two rules: the former one is used only when the type of the abstracted variable is an implicit.

Refine-new-lambda-implicit

$$\frac{\begin{array}{c}\mathcal{P},\ \Sigma,\ \Gamma \vdash ?\ \overset{\mathcal{R}}{\leadsto}\ ?_j : ?^s_k,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma; x :?_j \vdash M\ \overset{\mathcal{R}}{\leadsto}\ M' :\ T,\ \mathcal{P}'',\ \Sigma''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \lambda x :?.M\ \overset{\mathcal{R}}{\leadsto}\ \lambda x :?_j.M' :\ \Pi x :?_j.T,\ \mathcal{P}'',\ \Sigma''}$$

Refine-new-lambda-explicit

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{\mathcal{R}}{\leadsto} T_1' :\ s,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}'' = \mathcal{P}' \wedge \Gamma \vdash ?_k^s : \mathit{Type} \wedge \Gamma \vdash ?_j^c : s \to ?_k^s \\ \mathcal{P}'',\ \Sigma',\ \Gamma; x :?_i^c\ T_1' \vdash M \overset{\mathcal{R}}{\leadsto} M' :\ T,\ \mathcal{P}''',\ \Sigma''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \lambda x : T_1.M \overset{\mathcal{R}}{\leadsto} \lambda x :?_j^c\ T_1'.M' :\ \Pi x :?_j^c\ T_1'.T,\ \mathcal{P}''',\ \Sigma''}$$

The rule **Refine-prod** is replaced with the following two rules, the former one is applied only when the type of the quantified variable is an implicit.

Refine-new-prod-implicit

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash ? \overset{\mathcal{R}}{\leadsto} ?_j : ?_k^s,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma; x :?_j \vdash T \overset{\mathcal{R}}{\leadsto} T' :\ s_2,\ \mathcal{P}'',\ \Sigma'' \\ \mathcal{P}'',\ \Sigma'',\ \Gamma; x :?_j \vdash T' : s_2 \overset{?}{\equiv} \mathit{SortClass} \overset{\mathcal{C}}{\leadsto} T'',\ \mathcal{P}''',\ \Sigma''' \\ s_3 = \text{Obtained using the PTS rule on the type of } ?j \text{ and the type of } T''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \Pi x :?.T \overset{\mathcal{R}}{\leadsto} \Pi x :?_j.T' :\ s_3,\ \mathcal{P}'''',\ \Sigma''''}$$

Refine-new-prod-explicit

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma \vdash T_1 \overset{\mathcal{R}}{\leadsto} T_1' :\ s_1,\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}'' = \mathcal{P}' \wedge \Gamma \vdash ?_k^s : \mathit{Type} \wedge \Gamma \vdash ?_j^c : s_1 \to ?_k^s \\ \mathcal{P}'',\ \Sigma',\ \Gamma; x :?_j^c\ T_1' \vdash T_2 \overset{\mathcal{R}}{\leadsto} T_2' :\ s_2,\ \mathcal{P}''',\ \Sigma'' \\ \mathcal{P}''',\ \Sigma'',\ \Gamma; x :?_j^c\ T_1' \vdash T_2' : s_2 \overset{?}{\equiv} \mathit{SortClass} \overset{\mathcal{C}}{\leadsto} T_2'',\ \mathcal{P}'''',\ \Sigma''' \\ s_3 = \text{Obtained using the PTS rule on the type of } ?j \text{ and the type of } T_2''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash \Pi x : T_1.T_2 \overset{\mathcal{R}}{\leadsto} \Pi x :?_j^c\ T_1'.T_2'' :\ s_3,\ \mathcal{P}'''',\ \Sigma'''}$$

The rules for application are the original **Refine-appl-base** and **Refine-appl-rec** in which $\overset{\mathcal{C}}{\leadsto}$ is replaced with $\overset{\mathcal{C}_{\mathcal{LUB}}}{\leadsto}$.

Coerce-to-something-lub

$$\frac{\begin{array}{l}\mathcal{P},\ \Gamma \vdash T_1 \rightarrowtail T_2 \overset{\Delta_{\mathcal{LUB}}}{\leadsto} k,\ c\ ?1\ \ldots\ ?k\ \ldots\ ?n,\ \mathcal{P}' \\ \mathcal{P}',\ \Sigma,\ \Gamma \vdash ?k \overset{?}{\equiv} M \overset{\mathcal{U}}{\leadsto} \mathcal{P}'',\ \Sigma' \\ \mathcal{P}'',\ \Sigma',\ \Gamma \vdash c\ ?1\ \ldots\ ?k\ \ldots\ ?n \overset{\mathcal{R}}{\leadsto} M' :\ T_2',\ \mathcal{P}''',\ \Sigma'' \\ \mathcal{P}''',\ \Sigma'',\ \Gamma \vdash T_2 \overset{?}{\equiv} T_2' \overset{\mathcal{U}}{\leadsto} \mathcal{P}'''',\ \Sigma'''\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash M : T_1 \overset{?}{\equiv} T_2 \overset{\mathcal{C}_{\mathcal{LUB}}}{\leadsto} M',\ \mathcal{P}'''',\ \Sigma'''}$$

The auxiliary function $\overset{\Delta_{\mathcal{LUB}}}{\leadsto}$ returns the same informations that $\overset{\Delta}{\leadsto}$ returns: an index k, a term $c\ ?1\ \ldots\ ?k\ \ldots\ ?n$ and a new $\mathcal{P}$ such that the type T_2' is the least upper bound of all solutions in the coercion graph.

In case we are casting to $FunClass$ (the head of the application) the term T_2' must be of the form $\Pi x : A.B$ and among all possible products $\overset{\Delta_{\mathcal{LUB}}}{\leadsto}$ choses the least upper bound. Note that, according to the restrictions we made, the type of the head of the application cannot be a flexible term. Thus the computation of the least upper bound is as in Chen (that actually refers to the work of Aspinall and Compagnoni [6] for the $\mathcal{LUB}$ calculation).

In case we are casting to a type (the arguments of an application) $\overset{\mathcal{C}_{\mathcal{LUB}}}{\leadsto}$ the term $c\ ?1\ \ldots\ ?(k-1)$ must be such that the type $\Pi x : T_1'.\Pi x : T_{k+1}.\ldots\Pi x : T_n.T_2'$ is the least upper bound of the solutions in the coercion graph. The $\overset{\mathcal{C}_{\mathcal{LUB}}}{\leadsto}$ function is defined according to the restriction on functions in the algebraic language. Indeed, by hypothesis we must only consider the following two cases corresponding to the two kind of functions f in our language, and the argument a:

1. f has type $S \to T$ for some structure type S and some type T and a has type $?_1$ or it has type R for some structure type R. In the first case the $\mathcal{LUB}$ is the identity coercion and $?_1$ is unified with S. In the second case the $\mathcal{LUB}$ is the coercion from R to S in the coercion graph.

2. f has type $?_1.CA_R \to T$ and a has type $?_2$ or it has type $?_2.CA_S$. In both cases the $\mathcal{LUB}$ is the identity coercion and the type of a is unified with $?_1.CA_R$ exploiting the coercion graph as explained in Section 5.3.4.

Finally, as expected, our rules are not complete outside the fragment we choose. For instance, assume a coercion from natural numbers to integers and consider the following statement:

```
lemma broken : ∀f : (∀A : Type. A →A → Prop). f ?j 3 -2 ∧f ?j -2 3.
```

Here the type of f is completely specified, and the rule **Refine-new-prod-explicit** is applied. The term f ?j, which is outside our fragment, has type ?j → (?j → Prop) and it is passed an argument of type nat the first time and an argument of type int the second time. No backtracking free algorithm would be able to type this term.

5.3.6 More on the "with" construct

The **with** construct presented in Section 5.3.3 was only applied to the outermost structure. Moreover only one **with** constructor was present. Here we give a brief overview on how deep **with** constructs as well as nested **with** constructs can be implemented. We conclude with come considerations on **with** commutation that is not valid with our constructions.

Deep "with" construction

In order to interpret the **with** type constructor on "deep" fields, it is sufficient to follow the same schema, changing the coercion to make their sub-records manifest. Formally, when Q := {f: T; l: U} and R := {q: Q; s: S}, we interpret R **with** q.f = M with Σ c:R. c.q.f = M and we declare the coercion:

definition $\mathrm{k}_{\mathrm{R}}^{\mathrm{q.f}}$: ∀M:T. R **with** q.f = M → R :=
  λM:T. λx:(Σc:R. c.q.f=M).
    k_{R}^{q} (k_{Q}^{f} M $\langle \mathrm{x.1.1,\ x.2}\rangle_{\mathrm{Q},\lambda \mathrm{q:Q.q.f=M}}$)
      (**match** x **with** $\langle\langle\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}},\ \mathrm{s}\rangle_{\mathrm{R}},\ \mathrm{w}\rangle_{\mathrm{R},\lambda \mathrm{r:R.r.q.f=M}}$ ⇒
        $\langle\langle\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}},\ \mathrm{s}\rangle_{\mathrm{R}},\ [[\mathrm{refl}_{\mathrm{Q}}\ \langle \mathrm{a,\ l}\rangle_{\mathrm{Q}}]]_{\mathrm{w}}\rangle_{\mathrm{R},\lambda \mathrm{r:R.r.q=k_Q^f\ M\ \langle\langle a,l\rangle_Q,\ w\rangle_{Q,\lambda q:Q.q.f=M}}}$)

Note that the computational rule associated to the computational dependent elimination of Leibniz equality is necessary to type the previous coercion:

$$\langle\langle\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}},\ \mathrm{s}\rangle_{\mathrm{R}},\ [[\mathrm{refl}_{\mathrm{Q}}\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}}]]_{\mathrm{w}}\rangle_{\mathrm{R},\lambda \mathrm{r:R.r.q=k_Q^f M\langle\langle a,l\rangle_Q,w\rangle_{Q,\lambda q:Q.q.f=M}}}$$

is well typed since $\mathrm{refl}_{\mathrm{Q}}\ \langle \mathrm{a,\ l}\rangle_{\mathrm{Q}}$ has type $\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}} = \langle \mathrm{a,\ l}\rangle_{\mathrm{Q}}$ that is equivalent to

$$\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}} = \mathrm{k}_{Q}^{f}\ \mathrm{a}\ \langle\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}},\ \mathrm{refl_T\ a}\rangle_{\mathrm{Q},\lambda \mathrm{q:Q.q.f=q.f}}$$

Thus $[[\ \mathrm{refl}_{\mathrm{Q}}\ \langle \mathrm{a,\ l}\rangle_{\mathrm{Q}}]]_{w}$ has type $\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}} = \mathrm{k}_{Q}^{f}\ \mathrm{M}\ \langle\langle \mathrm{a,\ l}\rangle_{\mathrm{Q}},\ \mathrm{w}\rangle_{\mathrm{Q},\lambda \mathrm{q:Q.q.f=M}}$.

As expected, ($\mathrm{k}_{\mathrm{R}}^{\mathrm{q.f}}$M c).q.f ▷ M for all c of type R **with** q.f = M.

Nested "with" constructions

Finally, from the derived typing and reduction rules it is not evident that a type R **with** l_a=M **with** l_b=N can be formed. Surprisingly, this type poses no additional problem. The system simply de-sugars it as

$$\Sigma\, d\colon\ (\Sigma\, c{:}R.c.l_a{=}M).\ (k_R^{l_a}\ M\ d).l_b{=}N$$

and, as explained in the next section, automatically declares the composite coercion $k_R^{l_a,l_b}{:=}\ \lambda\, M,N,c.k_R^{l_b}\ N\ (k_R^{l_a}\ M\ c)$ as a coercion from R **with** l_a=M **with** l_b=N to R such that: $(k_R^{l_a,l_b}\ M\ N\ c).l_a{\rhd}M$ and $(k_R^{l_a,l_b}\ M\ N\ c).l_b{\rhd}N$ and

$$(k_R^{l_a,l_b}\ M\ N\ \langle\langle\langle M_1,\ldots,M_n\rangle_R,\ w_a\rangle,\ w_b\rangle).l_i{\rhd}\{\{M_i\}\}_{w_a,w_b}$$

where $\{\{M_i\}\}_{w_a,w_b}$ is M_i (if i<a and i<b), $[[\,M_i]]_{w_a}$ (if $a < i < b$), $[[M_i]]_{w_b}$ (if $b < i < a$), $[[[[\,M_i]]_{w_a}]]_{w_b}$ (if $a < b < i$ or $b < a < i$).

Signature strengthening and "with" commutation

To conclude our investigation of record types with manifest fields in type theory, we consider a few additional properties, which are *signature strengthening* and **with** commutation.

An important typing rule for dependently typed records with manifest fields is signature strengthening: a record c of type R must also have type R **with** f = R.f and the other way around. In our setting R **with** f = R.f is interpreted as $\Sigma\, c{:}R.\ c.f = c.f$ and we can couple the coercion k_R^f from R **with** f = R.f to R with a dual coercion ι_R from R to R **with** f = R.f such that: $\forall\, w.\ k_R^f(\iota_R(w)) \equiv w$, $\forall\, w.\iota_R(k_R^f(w)) = w$ and the latter Leibniz equality is strengthened to definitional equality when w.2 is a closed term or the system is proof irrelevant. The same can be achieved with minor modifications when the equality on the type of the f field is decidable.

with commutation is the rule that states the definitional equality of R **with** f=M **with** g=N and R **with** g=N **with** f=M when both expressions are well-typed. In our interpretation, the two types are not convertible since they are represented by different nestings of Σ-types. Moreover, for any label l that follows f and g in R, the l projection of two canonical inhabitants of the two types built from the same terms are provable equal, but not definitionally equal: in the first case we obtain a term $[[[[\ M]]_{w_f}]]_{w_g}$ for some w_f and w_g, and in the second case we obtain a term $[[[[\ M]]_{w_g}]]_{w_f}$. A proof of their equality is simply $[[[[\ \mathrm{refl}_T\ M]]_{w_f}]]_{w_g}$. Definitional equality holds when w_f or w_g are canonical terms — in particular when they are closed terms — or if at least one of the two types has a decidable equality. In practice, **with** commutation can often be avoided declaring a pair of mutual coercions between the two commutated types.

5.3.7 Remarks on code extraction

Algebra has been a remarkable testing area for code extraction, see the constructive proof developed in [34] for example. The encoding of manifest fields presented in the previous sections behave nicely with respect to code extraction. The manifest part of a term is encoded in the equality proof, that is erased during extraction, projections like k^f_R, are extracted to functions that simply replace one field of the record in input. All occurrences of $[[\ \_]]\ \_$ are also erased.

5.4 Coercions to sub-types

A recent work by Matthieu Sozeau [87] describes a language, Russell, that can be used to specify software. That language allows the user to write programs inside COQ using simple types, as if it was an ML program, and a specification using dependent types. Is up to Russell to generate the proof obligations needed to prove that the program satisfies the specification. For example look at the following program written in MATITA.

```
definition find :=
 λ p : nat → bool.
 let rec aux (l : list nat) :=
   λ l : list nat. match l with [ nil ⇒None | cons x tl ⇒
     match p x with [ true ⇒Some x | false ⇒aux tl]]
```

```
  in aux
```

The program find takes in input a test function p and returns the first element of the list l such that p l = true. It is essentially an ML program and use of complex types is avoided. The type of find is (nat → bool) → list nat → option nat. A possible specification for that program is the following

```
definition find_spec :=
 λ p,l.λ res:option nat.
  match res with
   [ None ⇒∀ y. mem ? eqb y l = true →p y = false
   | Some x ⇒mem ? eqb x l = true ∧p x = true ∧
       ∀ y.mem ? eqb y l = true → p y = true → x ≠ y →
          ∃ l1,l2,l3. l = l1 @ [x] @ l2 @ [y] @ l3.
```

The constant mem tests the natural number comparison function eqb partially applied to y, in the first case and x in the second case, against all elements in l. The specification states that, if the result of find is None no elements of the list satisfy p; if the result is an x then it is the fist element of the list satisfying p. We could have written find with a type like

```
∀ p:nat → bool.∀ l: list nat. { o : option nat | find_spec p l o}
```

But a program like that one needs all proof obligations concerning its return type to be proved inside the program code. Then, being irrelevant for the computation, a code extraction algorithm (or even a compiler) would have discharged all this information to obtain an efficient program. The approach proposed by Sozeau (already successfully exploited in PVS [75]) is to separate (at least to the users eyes) the code and the proof obligations, allowing him to write programs using a simple fragment of the functional language of CIC and to specify it using the full power of dependent types.

This approach is strictly related with coercions, especially with the flexible flavour implemented in MATITA. Consider the coercion tosigma that injects a term x of types option nat to Σ x : option nat, P x letting open then conjecture P x. This coercion can not be directly applied since our term (the find program) has type (nat $\rightarrow$ bool) $\rightarrow$ list nat $\rightarrow$ option nat.

If the tosigma coercion is propagated to the three distinct leaves of the find program (None, Some x and aux tl) then it will be applicable, since they both have type option nat. Moreover this propagation allows the user to attack proof obligations in the right context:

- in the first case l = [] and find returns None, thus the specification is
  $\forall$y. mem ? eqb y [] = true $\rightarrow$ p y = false
- in the second case l = x :: tl and p x = true the specification to be proved is the conjunction of mem ? eqb x (x::tl) = true with p x = true and
  $\forall$y.mem ? eqb y l = true $\rightarrow$ p y = true $\rightarrow$ x$\neq$ y $\rightarrow$ $\exists$l1,l2,l3.l = l1@[x]@l2@[y]@l3.
  The first simplifies to eqb x x || mem ? eqb x tl = true. The fist and second obligations are trivial, the third can be solved providing an empty l1.
- in the third case the find algorithm returns the recursive call. After the insertion of the coercion the recursive call returns a term enriched with its property (the specification itself applied to the recursive call). This step intuitively corresponds to the inductive step of the proof.

In the next section the propagation of coercions under lambda abstraction, pattern matching and recursive definitions is described.

5.4.1 Coercions propagation

Here we describe the implementation of coercion propagation we made in the refiner of MATITA. The main motivation for implementing that is to better stress the usage of coercions. We never aimed at implementing the Russell language in its full complexity. Nevertheless we obtained a working prototype of such tool exploiting the flexibility of subset coercions we implemented.

Moving under λ

Consider a term t of type $A \to B$. In our previous example of the find function we where casting only the output, but in general the input can be coerced too. Consider thus two coercions $c : A' \to A$ and $d : B \to B'$. The term we want to obtain is $\lambda x : A'.d\ (t\ (c\ x)) : A' \to B'$. Note that, if t has the form of a lambda abstraction $\lambda y.b[y]$ a β-redex is generated and the resulting term $\lambda x : A'.d\ ((\lambda y.b)(c\ x))$ can be reduced to $\lambda x : A'.d\ b[y/c\ x]$ obtaining a single lambda abstraction (that is why this operation is called propagation under lambda abstraction).

In the more general case of dependent products we use the following rule

Coerce-to-something-prod

$$\frac{\begin{array}{l}\mathcal{P},\ \Sigma,\ \Gamma; x : A' \vdash x : A' \overset{?}{\equiv} A \overset{\mathcal{C}}{\leadsto} x',\ \mathcal{P}',\ \Sigma' \\ \mathcal{P}',\ \Sigma',\ \Gamma; x : A' \vdash t\ x' : B[x/x'] \overset{?}{\equiv} B' \overset{\mathcal{C}}{\leadsto} t',\ \mathcal{P}'',\ \Sigma'' \\ t'' = \lambda x : A'.t'\end{array}}{\mathcal{P},\ \Sigma,\ \Gamma \vdash t : \Pi x : A.B \overset{?}{\equiv} \Pi x : A'.B' \overset{\mathcal{C}}{\leadsto} t'',\ \mathcal{P}'',\ \Sigma''}$$

Note that the type B lives in a context where x has type A, thus to put it under a context where $x : A'$ we replace every free occurrence of x in B with x'.

Moving under recursive definitions

The idea is similar to the lambda abstraction but formal notation makes it too heavy, we thus give an informal explanation of the steps involved. Consider the following recursive block:

$$\begin{array}{l}\text{letrec } f_1(x_{1,1}{:}T_{1,1})\ldots(x_{1,p_1}{:}T_{1,p_1}){:}T_{1,p_1+1} \text{ on } l_1 := t_1 \text{ and } \ldots \\ \text{and } f_n(x_{n,1}{:}T_{n,1})\ldots(x_{n,p_n}{:}T_{n,p'_n}){:}T_n \text{ on } l_n := t_n \text{ in } f_j\end{array}$$

Its type is:

$$\Pi x_{j,1} : T_{j,1} \ldots x_{j,p_j}{:}T_{j,p_j}.T_j$$

The bodies of recursive functions t_i are usually refined in a context where all recursive function types appear and f_j has type $\Pi x_{j,1} : T_{j,1} \ldots x_{j,p_j} : T_{j,p_j}.T_j$ (we will refer to that type with inferred-type). This recursive block is then casted to the type $\Pi x_{j,1} : S_{j,1} \dots x_{j,p_j} : S_{j,p_j}.S_j$ (we will refer to that type with expected-type).

The first step is to generate a context in which all recursive function type appear but the variable f_j is associated to the expected-type.

$$\begin{aligned}\Gamma = f_1 &: \Pi x_{1,1} : T_{1,1} \dots x_{1,p_1} : T_{1,p_1}.T_1; \dots ;\\ f_j &: \Pi x_{j,1} : S_{j,1} \dots x_{j,p_j} : S_{j,p_j}.S_j; \dots ;\\ f_n &: \Pi x_{n,1} : T_{n,1} \dots x_{n,p_n} : T_{n,p_n}.T_n\end{aligned}$$

In that context the variable f_j is cast to the inferred-type (the same operation done for x in the rule for propagation under lambda abstraction). The result of this cast f'_j is replaced in the body of every recursive function in place of f_j. Then, the body of f_j is casted to the expected-type (eventually using the rule **Coerce-to-something-prod**).

Moving under pattern matching

We first start with the basic approach, then we will refine it a bit.

Consider the following match construct:

$$M = \begin{array}{l}\text{match } t \text{ in } I \text{ return } T\\ [k_1\ x_{1_1}\ \dots\ x_{p_1} \Rightarrow t_1 \mid \dots\\ \mid k_n\ x_{1_n}\ \dots\ x_{p_n} \Rightarrow t_n]\end{array}$$

Consider also its type and the following declarations regarding the inductive type I:

$$\begin{array}{l} l, r, (I : \Pi x_1 : P_1. \dots \Pi x_l : P_l.\Pi x_{l+1} : P_{l+1}. \dots \Pi x_{l+r} : P_{l+r}.s) \in E\\ T : \Pi x_{l+1} : P_{l+1}. \dots .\Pi x_{l+r} : P_{l+r}.\Pi x : I\ p_1\ \dots\ p_l\ x_{l+1}\ \dots\ x_{l+r}.Q\\ t : I\ p_1\ \dots\ p_l\ p_{l+1} \dots p_{l+r}\\ M : T\ p_{l+1} \dots p_{l+r}\ t\end{array}$$

To reuse our previous code to propagate the coercion under lambda abstractions we need to build the new typing function and the type of every branch. If the new type to which the pattern matching construct has to be casted is S, the following terms have to be constructed:

$$
\begin{aligned}
&T' = \Pi x_{l+1} : P_{l+1}. \ldots .\Pi x_{l+r} : P_{l+r}.\Pi x : I\ p_1\ \ldots\ p_l\ x_{l+1}\ \ldots\ x_{l+r}.S \\
&k_i : \Pi x_1 : P_1. \ldots .\Pi x_l : P_l.\Pi y_1 : P_{i,1}. \ldots .\Pi y_{p_i} : P_{i,p_i}.I\ p_1\ \ldots\ p_l\ p_{i,l+1}\ \ldots\ p_{i,l+r} \\
&T_i = \Pi y_1 : P_{i,1}. \ldots .\Pi y_{p_i} : P_{i,p_i}.T\ p_{i,l+1}\ \ldots\ p_{i,l+r}\ (k_i\ p_1\ \ldots\ p_l\ p_{i,l+1}\ \ldots\ p_{i,l+r}) \\
&T'_i = \Pi y_1 : P_{i,1}. \ldots .\Pi y_{p_i} : P_{i,p_i}.T'\ p_{i,l+1}\ \ldots\ p_{i,l+r}\ (k_i\ p_1\ \ldots\ p_l\ p_{i,l+1}\ \ldots\ p_{i,l+r})
\end{aligned}
$$

Then each t_i can be coerced from T_i to T'_i and the typing function can be substituted with T'.

This preliminary implementation already works and coercions (to Σ-types) are applied deep enough in the recursive function to present to the user the proper assertions.

Sadly, in practical cases it is not enough. Consider the following sequent:

```
l : list nat
H : ∀n.mem ? eqb n l = false
==============================================
l = []
```

Here proceeding by case analysis on l leads to two goals: [] = [] and ∀x,l1. x :: l1 = []. The second one is not provable, since in the hypothesis H the term l has not been replaced by x::l1. What the user does, is to generalise the goal w.r.t H before proceeding by case analysis. With the generalisation the resulting proof term is a pattern matching construct where every branch takes in input an additional parameter H, and all occurrences of l in its type will be then decomposed by the pattern matching construction in every branch.

Another possibility, more involving for the user, but easier to implement is to generalise the goal on the term eq_refl l and then on all occurrences of l except the first, obtaining:

```
∀l2 : list nat. l = l2 → l2 = []
```

Then case analysis is performed on l2, and every branch of the pattern matching will have an additional hypothesis linking the decomposed l2 to l. In our example, in the second branch, the additional hypothesis would be `l = x :: l1`, that is exactly what is needed to solve the goal by means of `H`.

In our proof of concept implementation we adopted that trick, thus every branch is enriched with an additional lambda abstraction for the equality and the whole pattern matching is applied to the reflexivity proof.

This is the case where our implementation is more distant from Russell. More to the point, when pattern matching is performed over a term having a parametric inductive type, Leibniz equality is not able to state the link between the matched term and its instances (since the resulting equality may result to be ill typed). The usual solution to that kind of problems is to use a weaker equality, that relates terms whose types are not convertible. Since we were not interested in a full implementation of Russell we decided to not support this case.

Chapter 6

Automation and interactiveness in Matita

A huge part of our PHD has been dedicated to the development of automatic tactics for Matita. Modern interactive theorem provers usually integrate many domain specific tactics that are almost automatic, like Coq omega and ring [48] tactics. In the following we will consider only general automation, like Coq auto or auto rewrite tactics, since we focused on non-domain specific automatic tactics. Moreover, many domain specific automatic tactic are complete decision procedures, thus many considerations we will make do not apply.

The main motivation to work on automatic tactics is that the lack of comfortable automation has always been perceived as a grave deficiency by interactive theorem prover users, especially when the conjecture to prove is trivial (e.g. it is solvable using a lemma part of the library). We already observed in Section 2.3.2 that most automatic tactics tend not to scale very well to large developments, due to their black box nature. Changes to definitions break proof scripts, and automation usually hides the real cause of breakage, making the already tedious operation of mending the proof script harder. Moreover, after closing a conjecture using automation no trace of the found proof is left in the proof script file. Re-executing the proof script requires to execute again the automatic tactic, and this may take some time.

Things are considered to be automatic when they work without user intervention. On the contrary interactiveness is usually conceived as a continuous exchange of information between the user and the software assistant. We believe that the benefits these two different approaches bring are hard to get if techniques belonging to the two areas are not used in synergy. Modern computers are fast, but the user's intuition is still far behind what a calculator can "argue" by brute force. On the contrary a computer has no competitors in the manipulation of a huge dataset (just think to search engine, that nowadays index more than a billion of pages and are still able to give relevant answers).

Technologies developed by the automatic theorem proving community, although they showed their effectiveness in many occasions [35, 86], can hardly be considered user/interaction friendly, again due to their black box nature. We believe the integration of technologies developed by the automatic theorem proving community with modern interactive theorem provers can be a fruitful and challenging research objective, if some user interaction related requirements are satisfied.

We identified some requirements we think are necessary for a fruitful usage of automation inside interactive tools, and we developed two tactics for the Matita interactive theorem prover that, to some extent, fulfil these requirements. The former tactic (auto paramodulation from now on) performs automatic rewriting according to the superposition calculus [69], and has performed reasonably well against the huge TPTP problem set. The latter tactic (auto) performs applicative reasoning and has been designed to allow great user interaction.

The requirements we identified are:

- Integration of the searching facilities with automatic tactics.

  Automatic theorem provers are usually run on a predefined (small) set of assumptions. Interactive theorem provers are tools to author a library of proved theorems. That library is potentially huge, written by different users possibly not sharing the same office. These users may not (and usually do not) know all the contents of the library. Moreover, the more a library grows, the harder it is to have a strong control over it.

  Matita integrates the Whelp [1] search engine, this facility must be employed to help the user benefit from the library of already proved theorems.

- Interactive theorem provers are authoring tools thus they must help the user to maintain his work in a good shape. We already described in Section 2.3.2 why automatic tactic were avoided by the Mathematical Components team: their black box behaviour badly interacts with huge developments, since scripts are harder to mend if they break because of automation. Moreover, proof searching is an expensive operation and every time a proof script is check it is performed again. No reusable trace of a previous successful proof search is left in the script file.

  In our vision, automation has to be integrated with the authoring tool in such a way that these problems do not grow to the point of making it completely unusable.

- Interactive theorem provers have an heterogeneous set of users, that ranges from shy students to brilliant "hackers" that know every detail of the system. Black box automation makes both kind of users fall in the same category, since they have no easy way to tune it.

  A novice user can not feed an automatic tactic with the set of lemmas that are relevant, because he is not aware of such lemmas. He may like a non obscure procedure that shows him a set of possibilities, letting him inspect all of them, step by step learning what can be used to solve a given problem. A trained user can as well benefit from a clean proof searching procedure if the ongoing status of the search can be effectively displayed. Her intuition can drive the procedure in a smarter way than what an heuristic can do.

  Automation has thus to give clean feedback to the user, and possibly be tuned by him on the fly.

From these observations we formulated some proposals we think can solve some of these issues and that we followed in the implementation of the two automatic tactics auto and auto paramodulation:

- Automatic proof searching procedures must produce not only proof objects but proof scripts. This allows a fast re-execution of the proof script since proof steps are recorded in the script itself. This also allows to quickly mend a proof script, making it clear where the automatic procedure fails. If, after the rework of a definition, automation is not able to solve the goal anymore, the user still has the old proof script and can mend it by hand, a usually quicker operation than finding a new proof from scratch.

- Automatic proof searching procedure must be able to use efficient searching facilities to find by themselves which lemmas are relevant, since the user can not be always supposed to know the library she is working with.

- Tactics may allow user intervention and inspection, thus internal data structures have to be designed in such a way that a snapshot of the current status can be effectively presented to the user. Moreover the searching procedure itself must be designed to allow user intervention. Many algorithms have a shape that allows user intervention, since at some point they perform a choice. They employ cute heuristics but the user must be allowed to tune them, possibly on the fly.

Our view of automation is clearly biased by the interactiveness of tools like MATITA. When we began our PHD there were already two prototypes of automatic tactics, one performing rewriting using a restriction of the superposition calculus and another one doing Prolog like proof search using the **hint** whelp query (see Section 4.3.1).

We worked on the rewriting tactic making it pretty efficient and we developed a procedure to build nice proof objects (and as a consequence nice proof scripts) starting from a lightweight proof trace left by the automatic tactic. This work ended with the publication of the paper “Higher order Proof Reconstruction from Paramodulation-Based Refutations: The Unit Equality Case” [4]. This tactic has been tuned for performances using the huge TPTP library of problems of automatic theorem provers. Although these problems, in all their variants, seem to be mostly artificial we found them an invaluable test suite, and the tactic, optimised for such use case, proved to be quite efficient even when tackling real life conjectures. When we worked on this tactic our view on automation and interactiveness was not complete and we did not put efforts in allowing the procedure to be user driven, but we believe that the given clause algorithm that the tactic adopts can easily be made interactive. We left that as a future work.

During the last year of our PHD we reworked the tactic prototype that performs backward proof search, designing its code in such a way that the user can drive the procedure interactively. Together with our advisor we designed a graphical interface (that will be detailed later, the curious reader can peek Figure 6.4) that allows the user to follow the computation of the procedure, eventually interrupting or driving it. This tactic produces proof scripts and can use the search engine WHELP to automatically find interesting lemmas.

This chapter is organised in three sections. In the former we analyse the problems concerning the cooperation of the search engine and the automatic tactics. We will then describe the auto paramodulation tactic and detail the procedure to obtain nice proof objects from a lightweight proof trace left by the proof search engine. This section is an extract of the paper [4] we authored together with Andrea Asperti. Our main contribution of this work is the procedure to reconstruct the proof, while the implementation of the tactic itself has been done in full collaboration. The third chapter describes the auto tactic, dedicating special attention to the infrastructure to allow the user drive the proof search. Again the tactic, and its long and tedious tuning, is a joint work with our advisor, while the reworked implementation that allows user intervention is our contribution. The graphical user interface to drive the tactic, although designed together with our advisor, is our contribution.

6.1 Automation and searching

Interactive theorem provers are tools made to create a library of certified theorems. The effort to formalise a piece of mathematics is well known to be big. The so called De Bruijn factor, comparing the length in lines of a pen and paper proof with its formalised counterpart, was found to be ten. For this reason many users will probably collaborate when formalising some non trivial results, possibly parallelising as much as possible the work.

For that reason we believe that who designs interactive theorem provers must take into account the fact that users will probably not know all the details of the formalisation they are working on. Moreover, one expects to base any new development on possibly huge libraries of already formalised results, making it really impossible for a single user to know all the available facts.

We thus believe that these systems must help the user in finding relevant facts, and that automatic tactics must work up to the knowledge of the library of existing formalised results.

Anyway, the bigger the library is, the lower the probability that a given fact is pertinent to the user development is. We thus distinguish a subset of the library, that we will call the local context. The local context is the part of the library the user is actually working on (i.e. the theorems proved in files he is actually editing or that he references directly) and has to be treated with special care, since we believe it contains theorems strictly related to what the user is doing. This part of the library is smaller, and fitting into the main memory, can be indexed using extremely efficient data structures like discrimination trees.

In the following two sections we describe how our automatic tactics do find relevant facts in the whole library (Section 6.1.1) and in the local context (Section 6.1.2). Last section is dedicated to the tuning we made as a consequence of our formalisation experience regarding algebraic structures described in Chapter 3.

6.1.1 Searching the library

The WHELP search engine has been described in [1] that we co-author. Our main contribution in the development of whelp has been the tuning of the queries and the reorganisation of the SQL table to obtain reasonably good performances. This work has been done at the very beginning of our PHD.

Libraries of formalised theorems can be huge. For example the library of the COQ interactive theorem prover (together with all its third party contributions) amounts to 40,000 theorems and definitions. In such a setting where the amount of data makes it impossible to work in main memory, relational database are the obvious choice. The metadata model we store in the database has been briefly described in Section 4.3.1 and amounts to a ternary relation $s\ \mathcal{R}^p\ t$ stating that an object s refers an object t at a given position p. A minimal set of positions is used to discriminate the hypotheses (Hyp), from the conclusion (Concl) and the proof (Proof) of a theorem (respectively, the type of the input parameters, the type of the result, and the body of a definition). Moreover, the hypothesis and in the conclusion the root position (Main-Hyp and Main-Concl, respectively) is distinguished from deeper positions (that, in a first order setting, essentially amounts to distinguish relational symbols from functional ones).

On that simple metadata model we developed two kind of queries, to retrieve facts that can used to rewrite a given term or whose conclusion has good chances to unify with a given term.

Searching for unifiables

We start defining the basic operation μ.

Definition 6.1 (Constants of (μ)) *Given a lambda term t defined by the syntax of Table 5.2.1, $\mu(t)$ is computed by recursion over t collecting all c (constants), I (inductive types) and k (inductive constructors).*

Note that constants, inductive types and inductive constructors are all represented by name (actually a path in the general form of a URL like `cic:/...`).

The operation μ is extended to contexts Γ and to proof problems $\mathcal{P}$. In the syntax of CIC presented in 5.2.1 there are no local definitions (let-in) for simplicity, but they are used in the implementation of MATITA. Local definitions (when appearing in a context) have a special treatment, if the sort of the defined term is Prop, its body is skipped, while constants appearing in its body are collected by μ if it has sort Type.

Given a term g we are interested in lemmas, available in the library, whose type $t_1 \to t_2 \to \cdots \to t_n \to t$ is such that there exists a substitution θ that unifies t with g.

A necessary condition for that, which provides a very efficient filtering of the solution space, is that the set of constants in t must be a subset of those in g. This is clearly an approximation, since in CIC reduction can unfold constants, but in practice this approach shown to be sufficient.

In terms of our metadata model, the problem consists to compute all s such that

$$\{x | s\ \mathcal{R}^p\ x \text{ for some p}\} \subseteq \mu(g) \tag{6.1}$$

Iterating that for every s is clearly too expensive. The previous condition clearly holds only if there is a subset of $\mu(g)$ whose cardinality is equal to the cardinality of $\{x|s\ \mathcal{R}^p\ x\}$ for some s (and p). Note that the cardinality of $\{x|s\ \mathcal{R}^p\ x\}$ can be precomputed for every s and that the cardinality of $\mu(g)$ is usually small. Thus the set of s satisfying the following condition con be computed by a relational database with a join

$$\bigwedge_{a\in\mu(g)} s\ \mathcal{R}^p\ a \wedge |\mu(g)| = |\{x|s\ \mathcal{R}^p\ x\}| \tag{6.2}$$

The condition 6.1 is not sufficient to ensure that lemmas in resulting set have a conclusion that unifies with the goal. An additional unification step is performed (for example in the hint query) to filter the results. On the contrary, when we are interested in using this procedure to retrieve interesting lemmas, excessively filtering them is dangerous since the goal they may be applied to is not exactly the one that is used to perform the query.

Searching for equations or lemmas

In the previous section we introduced the overall idea employed to find theorems whose conclusion can hopefully unify with a given term. Here we give more details regarding the actual implementation of such search.

In Table 6.1 we report the procedure to collect the signature of a goal i (i is the index of a metavariable in the proof problem $\mathcal{P}$).

$\mu(i) = \mu(\Gamma_i) \cup \mu(T) \quad \Gamma \vdash ?i : T \in \mathcal{P}$
$s = \mu(i) \cup \{\mu(T)\|t \in \mu(i) \wedge \vdash t : T\}$

Table 6.1: Signature of goal

Note that s is closed with respect to the constants appearing in its types (e.g. if the successor function is in s then the type Nat is also in s).

Predicate symbols are filtered out of s, building $s_{\mathcal{P}}$ and s_T (respectively the predicates subset of s and its complement). The predicate symbols are used to refine query 6.2 with the following condition, where $P \in s_{\mathcal{P}}$.

$$\{x| \ldots \wedge x \ \mathcal{R}^{Main-Concl} \ P\}$$

In case we are looking for equations, P is forced to be *eq* (the inductive type of Leibniz equality).

6.1.2 Searching the local context

The local context is a small subset of the library of probable interest for the user. It is defined by the user explicitly, with the `include` statement that ensures that a given part of the library is available before proceeding. Anyway this set of lemma has to be compactly represented in memory in a data structure that also offers good searching facilities.

Discrimination trees are a widely studied data structure that has been adopted in many automatic theorem provers like Waldmeister [56]. They are essentially a tree structure, sharing common prefixes. All terms having a constant c as the principal symbol will share that node in the tree, making the in memory representation extremely compact. Moreover operations like pattern matching or unification have a linear (in the size of the input term) approximation visiting the tree.

An implementation of such structure was available in Matita as part of the prototype of the tactic performing automatic rewriting. It is not a perfect discrimination tree, thus the result of a search in the tree is not a set of terms together with a substitution but a set of terms that have a good chance to unify (or be a generalisation) of the input term. A proper handling of variables (like occur check) in not performed while searching the tree. Moreover there is no ad-hoc treatment of commutative symbols or more advanced techniques like the compilation of the tree into an efficiently interpretable language like in [77].

We use that code, letting any performance related improvement as a future objective. Anyway this simple implementation is already pretty fast. The main issue of using such data structure is that it is meant for a first order language, while CIC is higher order.

Discrimination trees and higher order logic

Discrimination trees shown to be an extremely efficient data structure to handle huge set of terms [68] but have been developed in a setting where variables represent never appear in the head position of an application. While this is perfect for the first order setting of most automatic theorem provers, CIC terms can have (meta)variables in head position. Moreover function definition can appear inside terms, and is in general undecidable if two of them are the same. The prototype of the automatic tactic was born having in mind to treat only problems trivially embedded in the first order fragment, that is a simple fragment already capturing many interesting theorems.

Anyway, to fruitfully employ the discrimination tree data structure we had to write a simple embedding of CIC terms into a first order syntax suitable for the already available data structure.

The following data type has been used to represent first order terms.

```
type fo_node =
  Constant of uri | Bound of int | Variable | Proposition | Datatype | Dead
```

First order term are thus represented with list of fo_node. Dead represents terms that are never considered equal (that is Dead <> Dead). Sorts are collapsed to Proposition for Prop and Datatype for every Type(j). Metavariables are considered Variable (i.e. subterms that can be instantiated by the pattern matching or unification query). Bound (de Bruijn indexes) are variables bound in the context and are considered fixed, equals only to themselves like constants (that are uniquely identified with a name instead of a integer). All Variables are the same, since we are interested in an approximation of unification or pattern matching. A subsequent refinement of the result will prune false matches.

The following function performs the conversion from CIC terms to the corresponding fo_node list.

```
let fo_node_of_cic = function
  | Cic.Meta _ →Variable
  | Cic.Rel i →Bound i
  | Cic.Sort (Cic.Prop) →Proposition
  | Cic.Sort _ →Datatype
  | Cic.Const _ | Cic.MutInd _ | Cic.MutConstruct _ as t →Constant (uri_of_term t)
  | Cic.LetIn _ | Cic.Lambda _ | Cic.Prod _ | Cic.Cast _
  | Cic.MutCase _ | Cic.Fix _ | Cic.CoFix _ →Dead
  | Cic.Appl _ →assert false (* should not happen *)
in
let rec fo_term_of_cic_term = function
    | Cic.Appl (Cic.Meta _ as hd::_) →[fo_node_of_cic hd]
    | Cic.Appl l →List.fold_left (l t →l @ fo_term_of_cic_term t) [] l
    | t →[fo_node_of_cic t]
```

Higher order metavariables are collapsed together with their arguments to variables.

6.2 Equational reasoning

In this section we give a description of the automatic tactic auto paramodulation.

Although the tactic is based on a prototype that was already available when our PHD begun, the amount of work needed to obtain a fully working and efficient tactic amounts to slightly less than one year. This tactic, and all the work to make it comply with our interactiveness related requirements, is our contribution. This work ended with the publication of the paper "Higher order Proof Reconstruction from Paramodulation-Based Refutations: The Unit Equality Case" [4] we co-author with our advisor prof. Andrea Asperti. What follows is partially an extract of that paper.

6.2.1 Superposition rules

Paramodulation [69] is precisely the management of equality by means of rewriting: given a formula (clause) $P(s)$, and an equality $s = t$, we may conclude $P(t)$. What makes paramodulation a really effective tool is the possibility of suitably constraining rewriting in order to avoid redundant inferences without loosing completeness. This is done by requiring that rewriting always replace *big* terms by *smaller* ones, with respect to a special ordering relation $\succ$ among terms, that satisfies certain properties, called the *reduction ordering.* This restriction of the paramodulation rule is called *superposition.*

Equations are traditionally split in two groups: facts (positive literals) and goals (negative literals). We have two basic rules: superposition right and superposition left. Superposition right combines facts to generate new facts: it corresponds to a forward reasoning step. Superposition left combines a fact and a goal, generating a new goal: logically, it is a backward reasoning step, reducing a goal G to a new one G'. The fragment of proof that can be associated to this new goal G' is thus not a proof of G' , but a proof of G *depending* on proof of G' (i.e. a proof of $G' \vdash G$).

We shall use the following notation: an equational fact will have the shape $\vdash M : e$, meaning that M is a proof of e; an equational goal will have the shape $\alpha : e \vdash M : C$, meaning that in the proof M of C the goal e is still open, i.e. M may depend on α.

Given a term t we write $t|_p$ to denote the subterm of t at position p, and $t[r]_p$ for the term obtained from t replacing the subterm $t|_p$ with r. Given a substitution σ we write $t\sigma$ for the application of the substitution to the term, with the usual meaning.

The logical rules, decorated with proofs, are the following:

Superposition left

$$\frac{\vdash h : l =_A r \qquad \alpha : t =_B s \vdash M : C}{\beta : t[r]_p\sigma =_B s\sigma \vdash M\sigma[R/\alpha\sigma] : C\sigma}$$

if $\sigma = mgu(l, t|_p)$, $t|_p$ is not a variable, $l\sigma \succ r\sigma$ and $t\sigma \succ s\sigma$; and $R : t\sigma =_B s\sigma$

Superposition right

$$\frac{\vdash h : l =_A r \qquad \vdash k : t =_B s}{\vdash R : t[r]_p\sigma =_B s\sigma}$$

if $\sigma = mgu(l, t|_p)$, $t|_p$ is not a variable, $l\sigma \succ r\sigma$ and $t\sigma \succ s\sigma$; and
$R : t[r]_p\sigma =_B s\sigma$

Equality resolution

$$\frac{\alpha : t =_A s \vdash M : C}{\vdash M[R/\alpha] : C}$$

if there exists $\sigma = mgu(t, s)$ and $R : t\sigma = t\sigma$

All proofs generated by the application of the three rules, named R, are omitted here, and will be described in Section 6.2.3.
The main theorem is that, given a set of facts S, and a goal e, an instance e' of e is a logical consequence of S if and only if, starting from the trivial axiom $\alpha : e \vdash \alpha : e$ we may prove $\vdash M : e'$ (and in this case M is a correct proof term).

Simplification rules such as tautology elimination, subsumption and especially demodulation can be added to the systems, but they do not introduce major conceptual problems, and hence they will not be considered here.

6.2.2 Implementation

The automatic proof search procedure is a component of MATITA, but is essentially orthogonal to the rest of the system.

CIC terms are translated into first order terms by a forgetful procedure that simply erases all type information, and transforms into opaque constants all terms not belonging to the first order framework as explained in 6.1.2

The inverse transformation takes advantage by the so called *refiner*, that is a type inference procedure typical of higher order interactive provers, detailed in Chapter 5.

Given the three superposition rules of Section 6.2.1, proof search is performed using the "given clause" algorithm (see [76, 78]). The algorithm keeps all known facts and goals split in two sets: active, and passive. At each iteration, the algorithm carefully chooses an equation (given clause) from the passive set; if it is a goal (and not an identity), then it is combined via superposition left with all active facts; if it is a fact, superposition right is used instead. The selected equation is added to the (suitable) active set, while all newly generated equations are added to the passive set, and the cycle is repeated.

As the reader may imagine a huge number of equations is generated during the proof search process, but only few of them will be actually used to prove the goal. Even if demodulation and subsumption are effective tools to discard equations without loosing completeness, all automatic theorem provers adopt clever techniques to strike down the space consumption of each equation. This usually leads to an extensive use of sharing in the data structures, and to drop the idea of carrying a complete proof representation in favour of recording a minimal and lightweight proof trace. The latter choice is usually not a big concern for ATP systems, since proofs are mainly used for debugging purposes, but for an interactive theorem prover that follows the independent verification principle like MATITA, proof objects are essential and thus it must be possible to reconstruct a complete proof object in CIC from the proof trace.

In our implementation the proof trace is composed by two slightly different kind of objects, corresponding to the two superposition steps. Superposition right steps are encoded with the following tuple, in OCaml syntax:

```
type rstep = ident * ident * direction * substitution * predicate
```

The two identifiers are unambiguous names for the equations involved (h and k in the former presentation of the superposition rule), *direction* can be either Left or Right, depending if h has been used left to right or right to left (i.e. if a symmetry step has to be kept into account). The *substitution* and the *predicate* are respectively the σ (i.e. the most general unifier between l and $t|_p$) and the predicate used to build the proof R, that is essentially a representation of the position $|_p$ identifying the subterm of t that has been rewritten with r once l and $t|_p$ were unified via σ.

This representation of the predicate is not optimal in terms of space consumption; we have chosen this representation mainly for simplicity, and left the implementation of a more compact coding as a future optimisation.

The representation of a superposition left step is essentially the same, but the second equation identifier has been removed, since it implicitly refers to the goal. We will call the type of these steps *lstep*.

A map $\Sigma : ident \rightarrow (pos\_literal * rstep)$ from identifiers to pairs of positive literal (i.e. something of the form $\vdash a =_A b$) and proof step represents all the forward reasoning performed during proof search, while a list Λ of *lstep* together with the initial goal (a negative literal) represent all backward reasoning steps.

Derivate tactics

The core of the auto paramodulation tactics implements many procedures that are of some interest even in other contexts.

For example, the demodulation simplification phase, directly connected with the search engine, is exported to the user under the name of demodulate tactic. When issued, it looks for interesting equations and tries to reduce the size of the goal (i.e. rewriting sub terms to smaller ones). The tactic never performs backtracking, simply rewrites until a minimum, according to the ordering relation between terms implemented. This minimum it thus not granted to be absolute. The intended usage it to support the simplify tactic, performing reduction steps internal to the logic. The usual example of a term the user may want to simplify but that is let untouched by the simplify tactic is $x + 0$ (when addition is defined by recursion on the first argument). The equation $\forall n.n+0 = n$ is like to be part of the standard library, and the hint rewrite query is able to find it. Demodulate reads this equation left to right (considering n less then $n + 0$) and rewrites with it. Note that other equations, like $\forall n, m.n + m = m + n$ are not properly oriented, thus are not used by demodulate.

Another derived tactic is smart apply. The apply tactic tries to unify the conclusion of a given lemma to the current goal. Going back to our previous example, if the goal is $P(n+0)$ and the lemma to be applied is $P(n)$ the user can not apply it directly (because unification works up to conversion and metavariable instantiation, not rewriting). Smart apply, when facing the previous problem, generates a goal $P(n) = P(n + 0)$ (that is an equation over propositions) and uses it to rewrite the current goal to the one the lemma is able to prove. Then auto paramodulation is used to solve the new goal. In that case the new goal can be easily proved rewriting the right hand side with the equation $\forall n.n + 0 = n$. This gives, to some extents, the illusion of having temporary extended the conversion rule with some (previously proved) equations.

6.2.3 Proof reconstruction

Here we address the problem of reconstructing a nice proof script from the proof trace left by the automatic procedure described in the previous section. First we need to introduce which CIC fragment will be used to represent such proofs, then we will describe the proof reconstruction procedure itself.

In the calculus of inductive constructions, equality is not a primitive notion, but it is defined as the smallest predicate containing (induced by) the reflexivity principle.

$$\text{Inductive eq } (A : \mathit{Type})\ (x : A) : A \to \mathit{Prop} \overset{\text{def}}{=} \text{refl\_eq} : \text{eq } A\ x\ x.$$

For the sake of readability we will use the notation $a_1 =_A a_2$ for (eq $A\ a_1\ a_2$).

As a consequence of this inductive definition, and similarly to all inductive types, it comes equipped with an elimination principle named eq_ind that, for any type A, any elements a_1, a_2 of A, any property P over A, given a proof h of $(P\ a_1)$ and a proof k that $a_1 =_A a_2$ gives back a proof of $(P\ a_2)$.

$$\frac{h : P\ a_1 \qquad k : a_1 =_A a_2}{(\text{eq\_ind } A\ a_1\ P\ h\ a_2\ k) : P\ a_2}$$

Similarly, we may define a higher order elimination principle eq_ind_r such that

$$\frac{h : P\ a_2 \qquad k : a_1 =_A a_2}{(\text{eq\_ind\_r } A\ a_2\ P\ h\ a_1\ k) : P\ a_1}$$

These are the building blocks of the proofs we will generate. With this definition of equality standard properties like reflexivity, symmetry and transitivity can be easily proved and are part of the standard library of lemmas available in MATITA.

We can now describe the R terms that were appearing in the proofs decorating the superposition rules.

Superposition left

$$\frac{\vdash h : l =_A r \qquad \alpha : t =_B s \vdash M : C}{\beta : t[r]_p\sigma =_B s\sigma \vdash M\sigma[R/\alpha\sigma] : C\sigma}$$

where $\sigma = mgu(l, t|_p)$ and $R = (\text{eq\_ind\_r}\ A\ r\sigma\ (\lambda x : A.t[x]_p =_B s)\sigma\ \beta\ l\sigma\ h\sigma) : t\sigma =_B s\sigma$

Superposition right

$$\frac{\vdash h : l =_A r \qquad \vdash k : t =_B s}{\vdash R : t[r]_p\sigma =_B s\sigma}$$

where $\sigma = mgu(l, t|_p)$ and $R = (\text{eq\_ind}\ A\ l\sigma\ (\lambda x : A.t[x]_p =_B s)\sigma\ k\sigma\ r\sigma\ h\sigma) : t[r]_p\sigma =_B s\sigma$

Equality resolution

$$\frac{\alpha : t =_A s \vdash M : C}{\vdash M[R/\alpha] : C}$$

where $\sigma = mgu(t, s)$ and $R = refl\_eq\ A\ t : t =_A t$.

The functions defined in Table 6.2 build a CIC proof term given the initial goal g, Σ and Λ. We use the syntax "let ($\vdash l =_A r,\ \pi_h$) = $\Sigma(h)$ in" for the irrefutable pattern matching construct "match $\Sigma(h)$ with ($\vdash$ eq $A\ l\ r$), $\pi_h \Rightarrow$".

The function ϕ produces proofs corresponding to application of the superposition right rule, with the exception that if h is used right to left and eq_ind_r is used to represent the hidden symmetry step. ψ builds proofs associated with the application of the superposition left rule, and fires ϕ to build the proof of the positive literal h involved.

Unfortunately this simple structurally recursive approach has the terrible behaviour of inlining the proofs of positive literals even if they are used non linearly. This may (and in practice does) trigger an exponential factor in the size of proof objects. The obtained proof object is thus of a poor value, because type checking it would require an unacceptable amount of time.

As an empirical demonstration of that fact we report in Figure 6.1 a graphical representation of the proof of problem GRP001-4 available in the TPTP[90] library version 3.1.1. Axioms are represented in squares, while positive literals have a circular shape. The goal is an hexagon.

```
$\phi(\Sigma,\ (h,\ k,\ dir,\ \sigma,\ P)) =$
    let $(\vdash l =_A r,\ \pi_h) = \Sigma(h)$ and $(\vdash t =_B s,\ \pi_k) = \Sigma(k)$ in
    match $dir$ with
    | Left $\Rightarrow$ eq_ind $A\ l\sigma\ P\sigma\ \phi(\Sigma,\ \pi_k)\sigma\ r\sigma\ \phi(\Sigma,\ \pi_h)\sigma$
    | Right $\Rightarrow$ eq_ind_r $A\ r\sigma\ P\sigma\ \phi(\Sigma,\ \pi_k)\sigma\ l\sigma\ \phi(\Sigma,\ \pi_h)\sigma$

$\psi'(\Sigma,\ (h,\ dir,\ \sigma,\ P),\ (t =_B s,\ \pi_g)) =$
    let $(\vdash l =_A r,\ \pi_h) = \Sigma(h)$ in
    match $dir$ with
    | Left $\Rightarrow (P\ r)\sigma,$ eq_ind $A\ l\sigma\ P\sigma\ \pi_g\sigma\ r\sigma\ \phi(\Sigma,\ \pi_h)\sigma$
    | Right $\Rightarrow (P\ l)\sigma,$ eq_ind_r $A\ r\sigma\ P\sigma\ \pi_g\sigma\ l\sigma\ \phi(\Sigma,\ \pi_h)\sigma$

$\psi(g,\ \Lambda,\ \Sigma) =$
    let $(t =_B s) \vdash \_ = g$ in
    $snd$(fold_right($\lambda x.\lambda y.\psi'(\Sigma,\ x,\ y),\ (t =_B s,$ refl_eq $A\ s),\ \Lambda$))

$\tau \overset{\text{def}}{=} term$
$\phi : (ident \rightarrow (pos\_literal * rstep)) * rstep \rightarrow \tau$
$\psi' : (ident \rightarrow (pos\_literal * rstep)) * lstep * (\tau * \tau) \rightarrow \tau$
$\psi : neg\_literal * lstep$ list $* (ident \rightarrow (pos\_literal * rstep)) \rightarrow \tau$
fold_right : $(lstep * (\tau * \tau) \rightarrow (\tau * \tau)) * (\tau * \tau) * lstep$ list $\rightarrow (\tau * \tau)$
```

Table 6.2: Proof reconstruction

Every positive literal points to the two used as hypothesis in the corresponding application of the superposition right rule. In this example a, b, c and e are constants, the latter has the identity properties (axiom H2). The thesis is that a group (axioms H3, H2) in which the square of each element is equal to the unit (axiom H1) is abelian (compose H with the goal to obtain the standard formulation of the abelian predicate). Equation 127 is used twice, 58 is used three times (two times by 127 and one by 123), consequently also 36 is not used linearly. In this scenario, the simple proof reconstruction algorithm inflates the proof term, replicating the literals marked

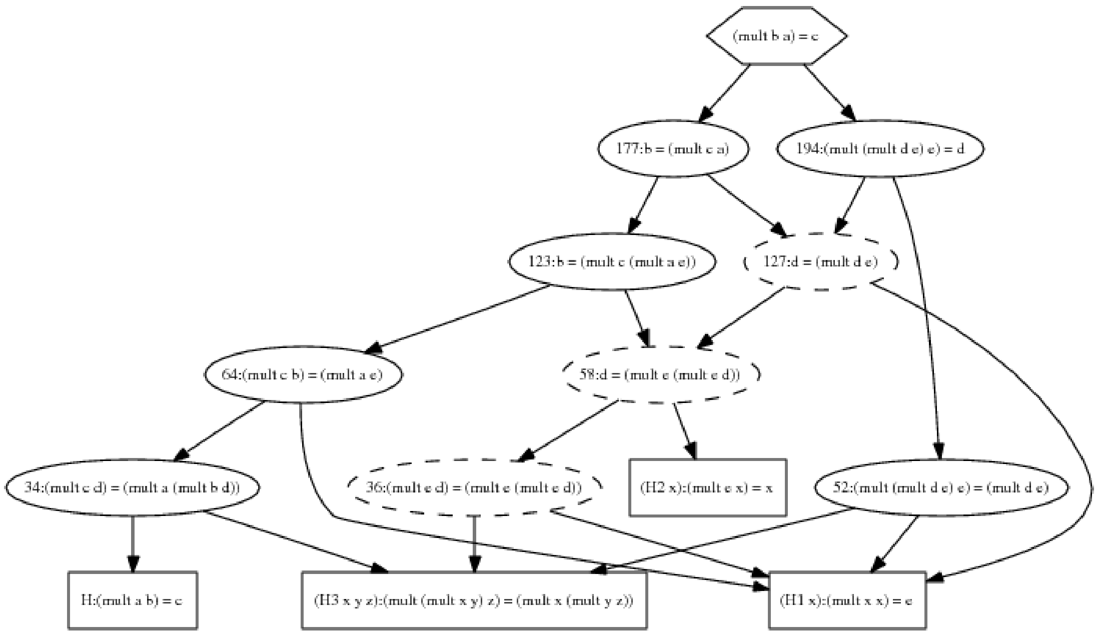

(mult b a) = c
177:b = (mult c a)
194:(mult (mult d e) e) = d
123:b = (mult c (mult a e))
127:d = (mult d e)
64:(mult c b) = (mult a e)
58:d = (mult e (mult e d))
34:(mult c d) = (mult a (mult b d))
36:(mult e d) = (mult e (mult e d))
(H2 x):(mult e x) = x
52:(mult (mult d e) e) = (mult d e)
H:(mult a b) = c
(H3 x y z):(mult (mult x y) z) = (mult x (mult y z))
(H1 x):(mult x x) = e

Figure 6.1: Proof representation (shared nodes)

with a dashed line.

The benchmarks reported in Table 6.3 show that this exponential behaviour makes proof objects practically intractable. The first column reports the time the automatic procedure spent in searching the proof, and the second one the number of iterations of the given clause algorithm needed to find a proof. The amount of time necessary to typecheck a non optimised proof is dramatically bigger then the time that is needed to find the proof. With the optimisation we describe in the following paragraph typechecking is as fast as proof search for easy problems like the ones shown in Table 6.3. As one would expect, when problems are more challenging, the time needed for typechecking the proof is negligible compared to the time needed to find the proof.

Fortunately CIC provides a construct for local definitions LetIn : $ident * term * term \rightarrow term$ that is type checked efficiently: the type of the body of the definition is computed once and then stored in the context used to type check the rest of the term.

We can thus write a function that, counting the number of occurrences of each equation, identifies the proofs that have to be factored out. In Table 6.4 the function γ returns a map from identifiers to integers. If this integer is greater than 1, then the corresponding equation will be factorised. In the example above, 127 and 58 should be factorised, since γ evaluates to two on them, and they must be factorised in this precise order, so that the proof of 127 can use the local definition of 58. The right order is the topological one, induced by the dependency relation shown in the graph.

Every occurrence of an equation may be used with a different substitution, that can instantiate free variables with different terms. Thus it is necessary to factorise closed proofs obtained λ-abstracting their free variables, and applying them to the same free variables where they occur before applying the local substitution. For example, given a proof π whose free variables are $x_1 \dots x_n$ respectively of type $T_1 \dots T_n$ we generate the following let in:

$$\text{LetIn } h \overset{\text{def}}{=} (\lambda x_1 : T_1, \dots \lambda x_n : T_n, \pi) \text{ in}$$

and the occurrences of π will look like $(h\ x_1\ \dots\ x_n)\sigma$ where σ will eventually differ.

Problem	Search	Steps	Typing		Proof size	
			raw	opt	raw	opt
BOO069-1	2.15	27	79.50	0.23	3.1M	29K
BOO071-1	2.23	27	203.03	0.22	5.4M	28K
GRP118-1	0.11	17	7.66	0.13	546K	21K
GRP485-1	0.17	47	323.35	0.23	5.1M	33K
LAT008-1	0.48	40	22.56	0.12	933K	19K
LCL115-2	0.81	52	24.42	0.29	1.1M	37K

Table 6.3: Timing (in seconds) and proof size

```
$\delta'(\Sigma,\ h,\ f) =$
    let $g = (\lambda x.\text{if } x = h \text{ then } 1 + f(x) \text{ else } f(x))$ in
    if $f(h) = 0$ then
        let $(\_,\ \pi_h) = \Sigma(h)$ in
        let $(k_1,\ k_2,\ \_,\ \_,\ \_) = \pi_h$ in
        $\delta'(\Sigma,\ k_1,\ \delta'(\Sigma,\ k_2,\ g))$
    else $g$

$\delta(\Sigma,\ (h,\ \_,\ \_,\ \_),\ f) = \delta'(\Sigma,\ h,\ f)$

$\gamma(\Lambda,\ \Sigma) = \text{fold\_right}(\lambda x.\lambda y.\delta(\Sigma,\ x,\ y),\ \lambda x.0,\ \Lambda)$

$\delta' : (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) * \mathit{ident} * (\mathit{ident} \to \mathit{int}) \to (\mathit{ident} \to \mathit{int})$
$\delta : (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) * \mathit{lstep} * (\mathit{ident} \to \mathit{int}) \to (\mathit{ident} \to \mathit{int})$
$\gamma : \mathit{lstep}\ \text{list} * (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) \to (\mathit{ident} \to \mathit{int})$
```

Table 6.4: Occurrence counting

Digression on dependent types and algebraic structures

The proof searching procedure we described operates in a first order setting, where all variables have the same type. CIC provides dependent types, meaning that in the previous example the type T_n can potentially depend on the variables $x_1 \ldots x_{n-1}$, thus the order in which free variables are abstracted is important and must be computed keeping dependencies into account.

Consider the case, really common in formalisations of algebraic structures, where a type, functions over that type and properties of these operations are packed together in a structure. For example, defining a group, one will probably end up having the following constants:

$$\begin{array}{ll}
\text{carr} : \mathit{Group} \to \mathit{Type} & \text{inv} : \forall g : \mathit{Group}, \text{carr } g \to \text{carr } g \\
\text{e} : \forall g : \mathit{Group}, \text{carr } g & \text{mul} : \forall g : \mathit{Group}, \text{carr } g \to \text{carr } g \to \text{carr } g \\
 & \text{id\_l} : \forall g : \mathit{Group}, \forall x : \text{carr } g, \text{mul } g \ (\text{e } g) \ x = x
\end{array}$$

Saturation rules work with non abstracted (binder free) equations, thus the id_l axiom is treated as $(mul\ x\ (e\ x)\ y = y)$ where x and y are free. If these free variables are blindly abstracted, an almost ill typed term can be obtained:

$$\lambda y : ?_1, \lambda x : ?_2, \text{mul } x \ (e \ x) \ y = y$$

where there is no term for $?_1$ such that $?_1 = (\text{carr } x)$ as required by the dependency in the type of mul: the second and third arguments must have type carr of the first argument. In the case above, the variable y has a type that depends on x, thus abstracting y first, makes it syntactically impossible for its type to depend on x. In other words $?_1$ misses x in its context.

When we decided to integrate automatic rewriting techniques like superposition in MATITA, we were attracted by their effectiveness and not in studying a generalisation of these techniques to a much more complex framework like CIC. The main, extremely practical, reason is that the portion of mathematical problems that can be tackled using first order techniques is non negligible and for some problems introduced by dependent types, like the one explained above, the solution is reasonably simple. Exploiting the explicit polymorphism of CIC, and the rigid structure of the proofs we build (i.e. nested application of eq_ind) it is possible to collect free variables that are used as types, inspecting the first arguments of eq_ind and eq: these variable are abstracted first. Even if this simple approach works pretty well in practice and covers the probably most frequent case of type dependency, it is not meant to scale up to the general case of dependent types, in which we are not interested.

6.2.4 Proof refinement

Proofs produced by paramodulation based techniques are very difficult to understand for a human. Although the single steps are logically trivial, the overall design of the proof is extremely difficult to grasp. This need is also perceived by the ATP community; for instance, in order to improve readability, the TPTP[90] library, provides a functionality to display proofs in a graphical form (called YuTV), pretty similar to the one in Fig. 6.1.

In the case of purely equational reasoning, mathematicians traditionally organise the proof as a chain of rewriting steps, each one justified by a simple side argument (an axiom, or an already proved lemma). Technically speaking, such a chain amounts to a composition of transitivity steps, where as proof leaves we only admit axioms (or their symmetric variants), possibly contextualized.

Formally, the basic components we need are provided by the following terms:

$$\begin{aligned}
&\text{trans} : \forall A : \mathit{Type}.\forall x, y, z : A.x =_A y \to y =_A z \to x =_A z\\
&\text{sym} : \forall A : \mathit{Type}.\forall x, y : A.x =_A y \to y =_A x\\
&\text{eq\_f} : \forall A, B : \mathit{Type}.\forall f : A \to B.\forall x, y : A.x =_A y \to (f\ x) =_B (f\ y)
\end{aligned}$$

The last term (function law) allows to contextualize the equation $x =_A y$ in an arbitrary context f.

The normal form for equational proofs we are interested in is described by the following grammar:

Definition 6.2 (Proof normal form)

$$\begin{array}{rcl}
\pi & = & \text{eq\_f}\ B\ C\ \Delta\ a\ b\ \mathit{axiom}\\
 & | & \text{eq\_f}\ B\ C\ \Delta\ a\ b\ (\text{sym}\ B\ b\ a\ \mathit{axiom})\\
 & | & \text{trans}\ A\ a\ b\ c\ \pi\ \pi
\end{array}$$

We now prove that any proof build by means of eq_ind and eq_ind_r may be transformed in the normal form of definition 6.2. The transformation is defined in two phases. In the first phase we replace all rewriting steps by means of applications of transitivity, symmetry and function law. In the second phase we propagate symmetries towards the leaves.

6.2.5 Phase 1: transitivity chain

The first phase of the transformation is defined by the ρ function of Table 6.5. We use Δ and Γ for contexts (i.e. unary functions). We write $\Gamma[a]$ for the application of Γ to a, that puts a in the context Γ, and $(\Delta \circ \Gamma)$ for the composition of contexts, so we have $(\Delta \circ \Gamma)[a] = \Delta[\Gamma[a]]$. The auxiliary function ρ' takes a context $\Delta : B \to C$, a proof of $(c =_B d)$ and returns a proof of $(\Delta[c] =_C \Delta[d])$.

$$
\begin{array}{l}
\rho(\pi) \leadsto \rho'(\lambda x{:}C.x,\ \pi) \qquad \text{when } \pi : a =_C b \\[1ex]
\rho'(\Delta,\ \text{eq\_ind } A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) \leadsto \\
\quad \text{trans } C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m] \\
\qquad (\text{sym } C\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta \circ \Gamma,\ \pi_2))\ \rho'(\Delta,\ \pi_1) \\[1ex]
\rho'(\Delta,\ \text{eq\_ind\_r } A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) \leadsto \\
\quad \text{trans } C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m]\ \rho'(\Delta \circ \Gamma,\ \pi_2)\ \rho'(\Delta,\ \pi_1) \\[1ex]
\rho'(\Delta,\ \text{eq\_ind } A\ a\ (\lambda x.m =_B \Gamma[x])\ \pi_2\ b\ \pi_1) \leadsto \\
\quad \text{trans } C\ \Delta[m]\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta,\ \pi_2)\ \rho'(\Delta \circ \Gamma,\ \pi_1) \\[1ex]
\rho'(\Delta,\ \text{eq\_ind\_r } A\ a\ (\lambda x.m =_B \Gamma[x])\ \pi_1\ b\ \pi_2) \leadsto \\
\quad \text{trans } C\ \Delta[m]\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b] \\
\qquad \rho'(\Delta,\ \pi_1)\ (\text{sym } C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \rho'(\Delta \circ \Gamma,\ \pi_2)) \\[1ex]
\rho'(\Delta,\ \pi) \leadsto \text{eq\_f } B\ C\ \Delta\ a\ b\ \pi \qquad \text{when } \pi : a =_B b \text{ and } \Delta : B \to C
\end{array}
$$

Table 6.5: Transitivity chain construction

In order to prove that ρ is type preserving, we proceed by induction on the size of the proof term, stating that if Δ is a context of type $B \to C$ and π is a term of type $a =_B b$, then $\rho'(\Delta,\ \pi) : \Delta[a] =_C \Delta[b]$.

Theorem 6.1 (ρ' injects) *For all B and C types, for all a and b of type B, if $\Delta : B \to C$ and $\pi : a =_B b$, then $\rho'(\Delta,\ \pi) : \Delta[a] =_C \Delta[b]$*

Proof: We proceed by induction on the size of the proof term.

Base case By hypothesis we know $\Delta : B \to C$, and $\pi : a =_B b$, thus a and b have type B and (eq_f $B\ C\ \Delta\ a\ b\ \pi$) is well typed, and proves $\Delta[a] =_C \Delta[b]$

Inductive case (We analyse only the first case, the others are similar)

By hypothesis we know $\Delta : B \to C$, and

$$\pi = (\text{eq\_ind } A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) : \Gamma[b] =_B m$$

From the type of eq_ind we can easily infer that $\pi_1 : \Gamma[a] =_B m$, $\pi_2 : a =_A b$, $\Gamma : A \to B$, $m : B$ and both a and b have type A. Since $\Delta : B \to C$, $\Delta \circ \Gamma$ is a context of type $A \to C$. Since π_2 is a subterm of π, by inductive hypothesis we have

$$\rho'(\Delta \circ \Gamma,\ \pi_2) : (\Delta \circ \Gamma)[a] =_C (\Delta \circ \Gamma)[b]$$

Since $(\Delta \circ \Gamma) : A \to C$ and a and b have type A, both $(\Delta \circ \Gamma)[a]$ and $(\Delta \circ \Gamma)[b]$ live in C. We can thus type the following application.

$$\pi_2' \overset{\text{def}}{=} (\text{sym } C\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta \circ \Gamma,\ \pi_2)) : (\Delta \circ \Gamma)[b] =_C (\Delta \circ \Gamma)[a]$$

We can apply the induction hypothesis also on $\pi_1' \overset{\text{def}}{=} (\rho'\ \Delta\ \pi_1)$ obtaining that is has type $(\Delta \circ \Gamma)[a] =_C \Delta[m]$. Since $\Delta[m] : C$, we can conclude that

$$\pi_3 \overset{\text{def}}{=} (\text{trans } C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m]\ \pi_2'\ \pi_1') : (\Delta \circ \Gamma)[b] =_C \Delta[m]$$

Expanding $\circ$ we obtain $\pi_3 : \Delta[\Gamma[b]] =_C \Delta[m]$

□

Corollary 6.1 (ρ is type preserving)

Proof: Trivial, since the initial context is the identity. □

$$\begin{aligned}
&\theta(\text{sym } A\ b\ a\ (\text{trans } A\ b\ c\ a\ \pi_1\ \pi_2)) \rightsquigarrow \\
&\qquad \text{trans } A\ a\ c\ b\ \theta(\text{sym } A\ c\ a\ \pi_2)\ \theta(\text{sym } A\ b\ c\ \pi_1) \\
&\theta(\text{sym } A\ b\ a\ (\text{sym } A\ a\ b\ \pi)) \rightsquigarrow \theta(\pi) \\
&\theta(\text{trans } A\ a\ b\ b\ \pi_1\ \pi_2) \rightsquigarrow \theta(\pi_1) \\
&\theta(\text{trans } A\ a\ a\ b\ \pi_1\ \pi_2) \rightsquigarrow \theta(\pi_2) \\
&\theta(\text{trans } A\ a\ c\ b\ \pi_1\ \pi_2) \rightsquigarrow \\
&\qquad \text{trans } A\ a\ c\ b\ \theta(\pi_1)\ \theta(\pi_2) \\
&\theta(\text{sym } B\ \Delta[a]\ \Delta[b]\ (\text{eq\_f } A\ B\ \Delta\ a\ b\ \pi)) \rightsquigarrow \\
&\qquad \text{eq\_f } A\ B\ \Delta\ b\ a\ (\text{sym } A\ a\ b\ \pi) \\
&\theta(\pi) \rightsquigarrow \pi
\end{aligned}$$

Table 6.6: Canonical form construction

6.2.6 Phase 2: symmetry step propagation

The second phase of the transformation is performed by the θ function in Table 6.6. The third and fourth case of the definition of θ are merely used to drop a redundant reflexivity step introduced by the equality resolution rule.

Theorem 6.2 (θ is type preserving) *For all A type, for all a and b of type A, if $\pi : a =_A b$, then $\theta(\pi) : a =_A b$*

Proof: We proceed by induction on the size of the proof term analysing the cases defining θ. By construction, the proof is made of nested applications of sym and trans; leaves are built with eq_f. The base case is the last one, where θ behaves as the identity and thus is type preserving. The following cases are part of the inductive step, thus we know by induction hypothesis that θ is type preserving on smaller terms.

First case By hypothesis we know that

$$(\text{sym } A\ b\ a\ (\text{trans } A\ b\ c\ a\ \pi_1\ \pi_2)) : a =_A b$$

thus $\pi_1 : b =_A c$ and $\pi_2 : c =_A a$. Consequently (sym $A\ c\ a\ \pi_2) : a =_A c$ and (sym $A\ b\ c\ \pi_1) : c =_A b$ and the induction hypothesis can be applied to them, obtaining θ(sym $A\ c\ a\ \pi_2) : a =_A c$ and θ(sym $A\ b\ c\ \pi_1) : c =_A b$. From that we obtain

$$(\text{trans}\ A\ a\ c\ b\ \theta(\text{sym}\ A\ c\ a\ \pi_2)\ \theta(\text{sym}\ A\ b\ c\ \pi_1)) : a =_A b$$

Second case We know that (sym $A\ b\ a$ (sym $A\ a\ b\ \pi)) : a =_A b$, thus (sym $A\ a\ b\ \pi) : b =_A a$ and $\pi : a =_A b$. Induction hypothesis suffices to prove $\theta(\pi) : a =_A b$

Third case Since (trans $A\ a\ b\ b\ \pi_1\ \pi_2) : a =_A b$ we have $\pi_1 : a =_A b$. Again, the induction hypothesis suffices to prove $\theta(\pi_1) : a =_A b$

Fourth case Analogous to the third case

Fifth case By hypothesis we know that

$$(\text{sym}\ B\ \Delta[a]\ \Delta[b]\ (\text{eq\_f}\ A\ B\ \Delta\ a\ b\ \pi)) : \Delta[b] =_B \Delta[a]$$

Thus $\pi : a =_A b$ and (eq_f $A\ B\ \Delta\ a\ b\ \pi) : \Delta[a] =_B \Delta[b]$. Hence (sym $A\ a\ b\ \pi) : b =_A a$ and

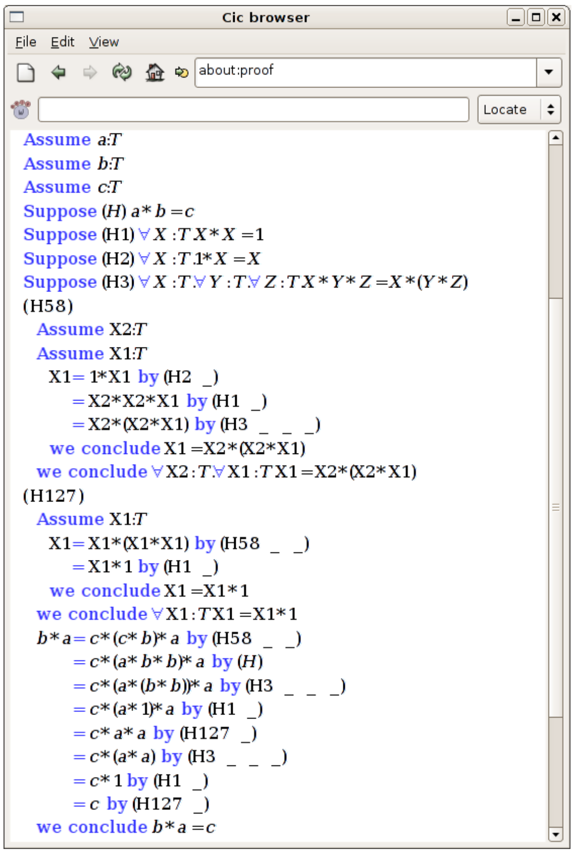
$$(\text{eq\_f}\ A\ B\ \Delta\ b\ a\ (\text{sym}\ A\ a\ b\ \pi)) : \Delta[b] =_B \Delta[a]$$

Sixth case Follows directly from the inductive hypothesis

□

6.2.7 Proof script generation

In Figure 6.2 we show an example of the kind of rendering obtained relative to the proof of GRP001-4 after all the previously described transformations are applied (i.e. the proof is in normal form).



```
Assume a:T
Assume b:T
Assume c:T
Suppose (H) a* b =c
Suppose (H1) ∀ X :T X * X =1
Suppose (H2) ∀ X :T.1* X =X
Suppose (H3) ∀ X :T.∀ Y :T.∀ Z :T X * Y * Z =X *(Y * Z)
(H58)
  Assume X2:T
  Assume X1:T
   X1= 1*X1 by (H2 _)
     =X2*X2*X1 by (H1 _)
     =X2*(X2*X1) by (H3 _ _ _)
   we conclude X1 =X2*(X2*X1)
  we conclude ∀X2 :T.∀X1 :T X1 =X2*(X2*X1)
(H127)
  Assume X1:T
   X1=X1*(X1*X1) by (H58 _ _)
     =X1*1 by (H1 _)
   we conclude X1 =X1*1
  we conclude ∀X1 :T X1 =X1*1
  b* a= c* (c* b)* a by (H58 _ _)
     = c* (a* b* b)* a by (H)
     = c* (a* (b* b))* a by (H3 _ _ _)
     = c* (a* 1)* a by (H1 _)
     = c* a* a by (H127 _)
     = c* (a* a) by (H3 _ _ _)
     = c* 1 by (H1 _)
     = c by (H127 _)
  we conclude b* a =c
```

Figure 6.2: Natural language rendering of the (refined) proof object of GRP001-4

The pretty printing facility used to generate that output was already part of MATITA and has been inherited from the MoWGLI project. The declarative language developed by Sacerdoti [26] is able to interpret that output, and the execution of the generated proof script produces the same proof object (the proof object we generated is the fix point of the composition of the pretty printing function and the execution function). The result is shown in Table 6.7.

In the implementation of the declarative language Sacerdoti used another tactic we implemented on top of the auto paramodulation core. The tactic, called solve rewriting, takes a goal, a set of lemmas (usually one) and a number of steps (usually one). It finds, using demodulation and backtracking, a solution of the goal (i.e. a rewrite sequence that leads to an identity) using only the given lemmas the number

```
theorem prove_b_times_a_is_c:
  ∀a,b,c : T.
  ∀H0 : a ∗ b = c.
  ∀H1 : ∀x:T. x ∗ x = 1.
  ∀H2 : ∀x:T. 1 ∗ x = x.
  ∀H3 : ∀x,y,z:T. (x ∗ y) ∗ z = x ∗ (y ∗ z).
  b ∗ a = c.
intros.
(∗∗ auto paramodulation. ∗)
  we need to prove (∀X2:T.∀X1:T.X1=X2∗(X2∗X1)) (H57).
  assume X2:T.
  assume X1:T.
   conclude X1= (1∗X1) by (H2 X1).
              = (X2∗X2∗X1) by (H1 X2).
              = (X2∗(X2∗X1)) by (H3 X2 X2 X1)
   done.
  we need to prove (∀X1:T.X1=X1∗1) (H125).
  assume X1:T.
   conclude X1= (X1∗(X1∗X1)) by (H57 X1 X1).
              = (X1∗1) by (H1 X1)
   done.
  conclude (b∗a)= (c∗(c∗b)∗a) by (H57 c b).
                = (c∗(a∗b∗b)∗a) by (H).
                = (c∗(a∗(b∗b))∗a) by (H3 a b b).
                = (c∗(a∗1)∗a) by (H1 b).
                = (c∗a∗a) by (H125 a).
                = (c∗(a∗a)) by (H3 c a a).
                = (c∗1) by (H1 a).
                = c by (H125 c)
  done.
(∗ end auto(Revision : 7586) proof: TIME=0.11 ∗)
qed.
```

Table 6.7: Proof script for GRP001-4

of times specified. The tactic is not meant for direct user consumption, but is the implementation of the declarative statements of the script. The lemma is specified on the right, but without the limitation given by the number of steps and the set of lemmas, calling the automatic tactic in its full power to solve the single equations may produce a different proof. This can mainly happen because the original input of the tactic were the first and the latter terms of the equality chain, intermediate step were proved in different context (e.g. having a different set of active equations).

6.2.8 The TPTP test case

We run the paramodulation tactic on 698 unit equality problems of the TPTP library version 3.2.0. The outcome is shown in the Appendix, in Table 8.1.

The number of problems successfully solved in less than ten minutes (the standard CASC timeout) is 510 on a relatively old Athlon microprocessor at 1.5 GHz. More than 350 of these problems were solved in less than ten seconds.

All the proofs found by the automatic tactic can be browsed online in the TPTP website, that runs MATITA on the same set of problems (with slightly better results since the hardware was faster). All proofs are successfully typechecked by the kernel of MATITA after the set of transformations described in the previous section.

These benchmarks are reported in the appendix and can be browser online at this url: `http://www.cs.miami.edu/~tptp/cgi-bin/DVTPTP2WWW/view_file.pl?Category=Problems`

6.3 Applicative reasoning

After the successful experience with auto paramodulation that proved to be reasonably fast and able to produce good proof scripts, we decided to put additional efforts on the interactiveness of automatic tactics.

The main motivations were that the debugging phase of the paramodulation tactic had been extremely difficult, mainly because the impossibility to easily drive the computation towards a case that was suspected of being buggy. Moreover the long tuning of the tactic running it on the TPTP library made clear that there is no good heuristic except the one that already knows on which goal the tactic will be called. Some automatic provers, like Waldmeister [56], provide the user a language to described patterns to which a particular set of options, meant to tune a prover, is associated. Almost every problem in the TPTP library has been classified in that language associating with it the best order relation over constants for example. We experimented many heuristics but the only result was to decrease the overall time to prove (or fail proving) all TPTP problems, but at the cost of continuous fluctuations of the time needed to prove every problem. We thought that being able to interactively drive the proof searching procedure would have made the debugging phase much simpler, and that it could have been of some use to the final user.

A prototype for a Prolog like proof searching procedure performing only backward reasoning steps was already available in MATITA, as a result of a master thesis. This tactic is essentially a recursive procedure performing backtracking: it tries to apply one of the lemmas an oracle provides, recursively solving newly generated goals. In case it fails, because the oracle produced no applicable lemmas or because a time (or step) limit is exceeded, backtracking is performed and the oracle is asked for an alternative solution (if any) for the previous problem.

The next section describes the algorithm and some optimisations, while Section 6.3.2 describes the data structure used to store all informations regarding the state of the algorithm. Section 6.3.3 describes the user interface and how the internal data structure described in Section 6.3.2 is processed to obtain a user friendly presentation of the tactic status. Last section describes how we produce a procedural proof script starting from the proof object generated by the tactic.

6.3.1 Backward reasoning

The computation performed by the algorithm can be represented as an and-or tree that is visited (and built) with a depth first policy. Figure 6.3 shows a sample.

Here G_1 is the initial goal, to which three lemmas were applicable. The applicable lemmas are alternatives (thus represent an or branching) respectively producing

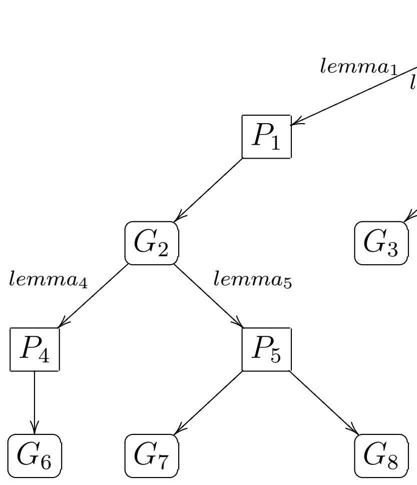


Figure 6.3: Sample and-or tree representing an auto computation

the proof problems P_1, P_2 and P_3. The former proof problem has only one open conjecture, while P_2 has two open conjectures. Both have to be solved to consider the application of the second lemma successful, thus this is an and branching. The depth first visiting policy already inspected G_2 that can be solved by means of two alternatives, respectively P_4 and P_5.

Note that if P_4 fails P_1 can still be solved successfully proving P_2. The same holds for G_1 that can not be considered not provable (by the tactic) until P_3 fails.

No automatic prover adopt this proof searching algorithm, mainly because the depth first policy that may lead to extremely long computations in the wrong direction. Anyway, the depth first policy is the one that better fits our needs. Even if the tree grows fast, the upper levels will probably be reasonably stable. The user has thus the time to look at what the prover is doing, possibly asking it to proceed on P_3 first for example. This intuition alone does not lead to interactiveness. The easiest way to write such proof searching algorithm is by recursion, having a decreasing parameter (for example the depth bound) to enforce termination. Not surprisingly, this recursion is not tail recursion, since we need the outcome of G_3 (for example) before proceeding on G_4. Thus, the information regarding the tree level n is available only in the n-th stack frame. OCaml does not allow to inspect the current call stack (only few interpreted language do). To give the user feedback of what is going on, we must be able to do snapshot of the current and-or tree. Maintaining this information in an imperative way is possible, but we considered it too error prone. A continuation passing style algorithm can do the job, but the higher order nature of the parameters has the same black-box nature of the call stack.

In addition to that requirement, the possibility of caching intermediate successes or failures has to be kept into account to obtain reasonable performances. The main difficulty in handling a cache is to have enough information to recognize a failure. For example, going back to the example in Figure 6.3, a failure for the intermediate goal G_2 can be cached when G_7 or G_8 fail (note that they can fail only after that G_6 failed, since the tree is visited using a depth first policy). On the contrary, a success (and a proof) for the intermediate goal G_2 can be cached as soon as G_6 is solved (or G_7 and G_8). Going back to the name we gave to that tree (and-or) is clear that a goal G can be considered solved when one of its children is solved (or branching) but can be considered unsolvable only when all of its children are failed (not or branching). Drawing a parallel with programming languages implementation, the former condition is computed as a lazy connective.

The data structure we adopted to describe a computation, allowing tail recursion (thus a stack-less iteration) but containing enough information to reconstruct the tree and properly handle caching is explained in the next section.

6.3.2 The data structure and the algorithm

The data structure we used is not straightforward, we thus first introduce it informally with an example, then we move to an operational description of the structure and its manipulation.

The simple idea that is at the base of the data structure is to project the tree of Figure 6.3 on its base. Every or branching generates a number of new list elements as its branching factor. Every subsequent and branching distributes over the previous (upper) or branching elements. The obtained list is the following:

$$(P_4, [G_6]) :: (P_5, [G_7; G_8]) :: (P_2, [G_3; G_4]) :: (P_3, [G_5]) :: []$$

Every element of that list can be considered a complete or branch (solving one of these elements leads to a solution of the initial problem). Note that the depth fist policy we use to build the tree needs to examine only the first element of the list, letting the tail unmodified. Moreover every element has its own proof problem, making it in some sense complete (no need to use information stored in other elements to process it).

This simple structure is not enough to reconstruct the tree and to perform caching, thus every element is enriched with more information. Every goal is marked with D (to do) or with S (save success). These marks are interpreted as instructions, if the first item in the goals list is a D then the goal is processed trying to solve it, if it is a S operations regarding caching are performed. Our example, enriched with this information becomes:

$$\begin{gathered}(P_4, [D_{G_6}; S_{G_2}; S_{G_1}]) :: \quad (P_5, [D_{G_7}; D_{G_8}; S_{G_2}; S_{G_1}]) :: \\ (P_2, [D_{G_3}; D_{G_4}; S_{G_1}]) :: \quad (P_3, [D_{G_5}; S_{G_1}]) :: []\end{gathered}$$

To understand how this structure works, we execute one step. The step will produce the just shown element list, starting from the previous state. Consider the tree where the subtree rooted in G_6 is removed. We build the same enriched structure and obtain:

$$(P_1, [D_{G_2}; S_{G_1}]) :: (P_2, [D_{G_3}; D_{G_4}; S_{G_1}]) :: (P_3, [D_{G_5}; S_{G_1}]) :: []$$

when processing the first element, it generates in one step the two head elements:

$$(P_4, [D_{G_6}; S_{G_2}; S_{G_1}]) \qquad (P_5, [D_{G_7}; D_{G_8}; S_{G_2}; S_{G_1}])$$

The item D_{G_2} (representing that G_2 has to be processed) can be solved with two lemmas, thus the element is duplicated and the processed item is replaced by a list of D ending with an S, with a D item for each newly opened goal. When the processing of a D item produces no new goals, the subsequent and corresponding S item is processed, saving in cache the proof found for that goal. Note that the proof problems P_i we considered here are enriched with some information (a substitution environment Σ) that allow to reconstruct the proof of every conjecture not appearing anymore open in the proof problem.

This additional information is necessary to properly cache successes, what is still missing is the possibility to cache failures. Every elements of the list is enriched with a third component listing the goals that has to be considered failed when the element (actually the head D item of the element) fails. This information is inherited by the rightmost element generated when an or branching is performed.

Operational description of the tactic

To describe formally how this structure is employed we need to define the following abstract objects.

Definition 6.3 (Cache) *A cache θ is a partial function from terms (actually types) to terms. Its domain can be extended with the operation $\theta[T \mapsto t]$. All terms in θ live in the same context.*

We also define the following function to extract from a substitution environment the proof a given goal.

Definition 6.4 (Proof of goal) *Given a goal (metavariable number) g and a substitution environment Σ, the proof of g denoted with $\Sigma(g)$ is the least fixed point of $\Sigma(\cdot)$ starting from $?g$.*

We use the notation $\theta[T \mapsto \Sigma(g)]$ to update θ associating the proof of g with T. We use $\perp$ to represent failures, thus $\theta[T \mapsto \perp]$ extends θ with the information that T has no proof.

Since $\mathcal{P}$ and Σ are always coupled, we call them P from now on (of type P).

Definition 6.5 (Element) *We call an element a tuple of type (in OCaml notation) $P * oplist * goallist$ where goal is the type of metavariable indexes and op is the algebraic type with the D and S constructors. D taking in input a goal, S a goal and a term.*

Finally, the function to find lemmas that can be applied to a given goal is *cands*.

Definition 6.6 (Candidates (of the environment E)) *Let g be a goal, $\mathcal{P}$ a proof problem and Σ a substitution environment. Let $\Gamma \vdash ?g : T \in \mathcal{P}$. $cands((\mathcal{P}, \Sigma), g)$ returns a list of elements $(t, (\mathcal{P}', \Sigma'), g_1 \dots g_n)$ such that:*

- $t \in E$
- $[] \vdash t : T_1 \to \dots \to T_n \to T'$
- $\mathcal{P},\ \Sigma,\ \Gamma \vdash T \overset{?}{\equiv} T' \overset{\mathcal{U}}{\leadsto} \mathcal{P}',\ \Sigma'$
- $\Sigma'(\Gamma) \vdash ?g_i : \Sigma'(T_i) \in \mathcal{P}'$

Note that *cands* can easily be extended to look for t not only in E but also in θ since all elements in θ live in the same context as T.

In Table 6.8 we define the *step* function mapping a list of elements to a new list of elements. This is the core of the auto tactic.

The functions *sort* and *purge* have not been defined, for the moment both can be considered to be the identity. The step function is not defined for the status $([], \theta)$ since it represent complete failure: the elems list can be considered to list all alternatives to prove the initial goal, being empty means that all alternatives have been explored with a negative result.

The annotation t in S_g^t is not used in the operational semantic, and t represents the lemma that was applied to g. Remember we have to show the user the history of lemmas applied so far (the lemmas that annotate the arrows in the tree of Figure 6.3). An element $((P, S^{t_1} :: \dots :: S^{t_n}, fl), \theta)$ represents a tree where all goals have been solved.

As we stated in the informal description of the algorithm, the procedure can be limited to a certain depth, and even a number of nodes. To efficiently keep track of the depth or size of the tree, the element structure is enriched with that information: every time a D item is processed, the depth limit (as well as the size) is decreased.

$$(((\mathcal{P},\Sigma) \text{ as } P, S_g^t :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el', \theta')$$
where $\theta' = \theta[T \mapsto \Sigma(g)]$ and $\Gamma \vdash ?g : T \in \mathcal{P}$ and $el' = purge(el)$
and when $\mathcal{M}(T) = \emptyset$

$$(((\mathcal{P},\Sigma) \text{ as } P, S_g^t :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el, \theta)$$
where $\Gamma \vdash ?g : T \in \mathcal{P}$
and when $\mathcal{M}(T) \neq \emptyset$

$$((P, D_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P'_1, l_1@tl, []) :: \ldots :: (P'_m, l_m@tl, g :: fl) :: el, \theta) \qquad (*)$$
where $cands(P,g) = (t_1, P'_1, g_{1,1} \ldots g_{1,n_i}) :: \ldots :: (t_m, P'_m, g_{m,1} :: \ldots :: g_{m,n_m})$
and $l_i = sort([D_{g_{i,1}} \ldots ; D_{g_{i,n_i}}]) :: [S_g^{t_i}]$ for $i \in \{1 \ldots m\}$

$$((P, D_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, [], fl) :: el, \theta')$$
when $cands(P,g) = []$

$$(((\mathcal{P},\Sigma) \text{ as } P, D_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, [], fl) :: el, \theta) \qquad (**)$$
when $\theta = \theta'[T \mapsto \bot]$ and $\Gamma \vdash ?g : T \in \mathcal{P}$

$$((P, [], fl) :: el, \theta) \xrightarrow{step} (el, \theta')$$
where $\theta' = \theta$ and $\Gamma_g \vdash ?g : T_g \in xP$ for $g \in fl$ and $\theta' = \theta[T_g \mapsto \bot]$ for $g \in fl$

$$([], \theta) \xrightarrow{step} \qquad \text{(Failure)}$$

Table 6.8: Auto tactic operational description

When an S item is processed the depth is increased again. The additional following rule is then added to the operational description:

$$((P, items, fl, depth, size) :: elems, \theta) \xrightarrow{step} ((P, [], fl, depth, size) :: elems, \theta)$$
when $depth < 0 \vee size < 0$

The cache θ is still not optimal, since a goal g of type T can be associated with $\bot$ because the algorithm run out of depth (or size). If the algorithm encounter again the same goal type T with a greater depth, it could retry. To fix this problem, goals have to be paired together with the depth at which they have been encountered in the failure (fl) list, and the $\bot$ symbol annotated with that depth in the rule for elements with an empty item list $(((P, [], fl), \theta))$. Then the rule (**) can be replaced with

$$(((\mathcal{P}, \Sigma) \text{ as } P, D_g :: tl, fl, depth, size) :: elems, \theta) \xrightarrow{step} ((P, [], fl, depth, size) :: elems, \theta)$$
$$\text{when } \theta = \theta'[T \mapsto \bot_n] \text{ and } \Gamma \vdash ?g : T \in \mathcal{P} \text{ and } depth \leq n$$

The *sort* function can be used to implement an heuristics to choose which alternative the next *step* computation should attack. The simplest heuristic is to count the number of newly generated goals (the length of l_i in the rule (*)).

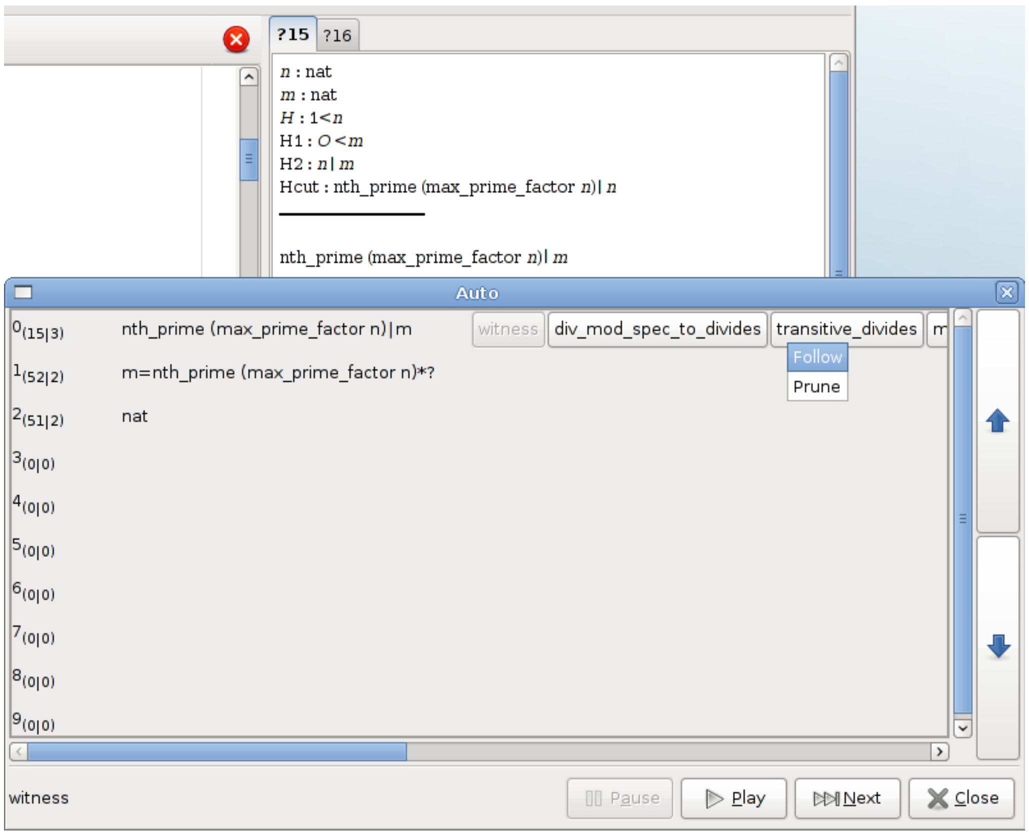
The *purge* function is more sophisticated and is used to drop alternatives (brothers in the tree). It is used only if the type of the goal is free of metavariables. If it contains metavariables, every different proof may instantiate them in a different way and that may make other goals false. In that case, we do not cache the solution.

If the goal type is metavariable free, we cache the proof and remove all its brothers. They can be identified (in the flat elem list) comparing the list of items: since the tl is inherited by all brothers (in rule (*)).

6.3.3 Driving automation

In Figure 6.4 the window to drive the automatic tactic is shown. On the background there is the main window of MATITA, showing the current open conjecture (conjecture fifteen). The window is divided in three columns:

- the leftmost shows the progressive number of open conjectures, the number of the conjecture examined and the depth left (with standard parameters auto has a bound of three);
- the column in the middle displays the current conjecture;
- the rightmost column lists all lemmas that can be applied to the conjecture.



```
?15  ?16

n : nat
m : nat
H : 1<n
H1 : O<m
H2 : n| m
Hcut : nth_prime (max_prime_factor n)| n
________________

nth_prime (max_prime_factor n)| m
```

```
Auto
0(15|3)   nth_prime (max_prime_factor n)|m       witness  div_mod_spec_to_divides  transitive_divides  m
1(52|2)   m=nth_prime (max_prime_factor n)*?                                       Follow
2(51|2)   nat                                                                      Prune
3(0|0)
4(0|0)
5(0|0)
6(0|0)
7(0|0)
8(0|0)
9(0|0)

witness
```

Figure 6.4: Auto interaction window

To attack conjecture fifteen the automatic tactic found a bunch of lemmas that can be applied. The former, witness, has already been applied and is thus coloured in gray. All its alternatives are shown on its right. The application of the witness lemma to a goal of the form $n|m$ opens two conjectures: the former (number 52) is that for a certain ?, $m = n*?$ and the latter (number 51) is the witness ? itself. The next step performed by auto is to find relevant lemmas for the conjecture displayed in the second line, place them in the rightmost column, gray the former and display the result of its application. In case one application fails, the next alternative is attempted. In case there are no alternatives left, the next alternative of the previous line it considered. Thus, if no lemmas can be applied to conjecture 52, both line one and two are removed together with the witness lemma that generated them and the lemma div_mod_spec_to_divides is applied.

The user can execute the tactic step by step with the next button, and switch between the running status and the paused one with the buttons pause and play. To drive the proof searching algorithm the user can interact with the lemmas in the rightmost column. In Figure 6.4 the user just clicked on the transitive_divides lemma, opening the list of allowed actions. The prune action simply removes the lemma for the list of alternatives, the follow action makes all alternatives before the one selected immediately fail.

Information reconstruction

The window shown in Figure 6.4 is able to render the content of the following data structure, that is almost isomorphic to what is displayed:

```
type auto_status =
  (int * Cic.term * int * (int * Cic.term) list ) list *
  (int * Cic.term * int) list
```

The fields of the first components are respectively the number of the goal, its type and the depth at which it is being processed. The int ∗ Cic.term list represents the alternatives for the goal, the term being the lemma, while the number is an identifier that can be used to drive the automation (following or dropping the alternative). The second component represents the list of goals of the first element (the one being processed). The first integer is the goal identifier, the term is its type and the last integer is the depth.

The procedure to generate such data starting from a list of elements is not of particular interest, but is reported for the sake of completeness.

The second component can be easily generated from the *oplist* component of the first element $(P, oplist, fl)$, filtering out all S elements. The first component is harder to build. The list of elements is projected dropping all but the *oplist* and filtering out items marked with D and adding the depth component to every new item. The obtained structure has thus the following type: (goal ∗ Cic.term ∗ int) list list.

This list is processed, in reverse order starting with an empty accumulator, with the following function:

```
let rec eat_all rows = function
  | [] →rows
  | elem:: or_list as l →
      match List.rev elem with
      | ((goal,cand,depth)::_ →
          let eaten, l = eat_in_parallel goal l in
          let rows = rows @ [goal, depth, List.rev eaten] in
          eat_all rows l
      | [] →eat_all rows or_list
in
  eat_all []
```

The idea is to reconstruct the tree from its root. Going back to Figure 6.3, the last element will be the one relative to P_3, $(P_3, [D_{G_5}; S_{G_1}^{lemma_3}])$, that has been preprocessed to $(G_5, lemma_3, depth)$. The function eat_in_parallel will thus remove all elements (starting from the tail) that have G_5 as the first component, collecting all the lemmas (the second component). The resulting row will thus be:

$$[G_5, depth, [lemma_1; lemma_2; lemma_3]]$$

The recursive call will then eat all elements having G_2 as the first component, collecting $lemma_4$ and $lemma_5$.

The utility function eat_in_parallel is reported below and simply eats the last element of each list accumulating some components of the eaten elements.

```
let eat_in_parallel id l =
  let rec aux (eaten, new_l as acc) = function
    | [] →acc
    | l :: tl →
        match eat_tail_if_eq id l with
        | None, l →aux (eaten, new_l@[l]) tl
        | Some t,l →aux (eaten@[t], new_l@[l]) tl
  in
    aux ([],[]) l
```

The utility function eat_tail_if_eq eats all elements having the first component equal to its first argument. Note that the lists in input are already reversed.

```
let eat_tail_if_eq id l =
  let rec aux (s, l) = function
    | [] →s, l
    | (id1,c,depth):: tl when id = id1 →
        (match s with None →aux (Some c,l) tl | Some _ →assert false)
    | (id1,c,depth):: tl →aux (s, e :: l) tl
  in
   let c, l = aux (None, []) l in c, List.rev l
```

6.3.4 Proof reconstruction

As we already mentioned in Chapter 4 the facility of producing a declarative proof script [26] starting from a proof object is now part of MATITA, but the procedure for producing a procedural proof scripts is not finished yet (and its implementation was not even in progress when we worked on the tactic). Since we were interested in obtaining a procedural proof script, that we believe more appealing for a computer scientist used to programming languages, we had to implement an had hoc procedure to obtain a reasonable proof script.

Since proofs are found with a simple backward reasoning algorithm, they have a rigid structure, essentially an initial spine of lambda abstractions and then nested applications. This structure corresponds to an application of the intros tactic and a sequence of properly nested call to the apply tactic.

The head of every application corresponds to an apply command, and every argument whose sort is Prop has to be processed recursively as a nested apply command.

What follows is the proof script generated by the auto tactic on the goal described in Section 6.3.3.

```
(** auto. *)
apply (transitive_divides (nth_prime (max_prime_factor n)) n m ? ?);
  [apply (Hcut).
  |apply (H2).
  ]
```

As one can see, the dependency of types to terms is not kept into account, since all the arguments of transitive_divides can be inferred from the successive applications. This kind of optimisation was out of our scope, since we were interested only in providing a working procedural script. Guidi in [49] develops a full featured procedural pretty printer for proof objects, that will be substituted to that proof of concept in the near future.

Chapter 7

Conclusions

In this thesis we describe our experiences and research results as a user and as a developer of interactive theorem provers.

In the first part of the thesis we describe two distinct formalisation experiences on two different interactive theorem provers, giving a detailed analysis of the methodologies we used and the issues we faced.

The first one describes the work we made during our internship in the Mathematical Components team leaded by Georges Gonthier. Our work consisted in the formalisations of some results of finite group theory, with particular care in the design of basic definitions and data types. We defined finite-intensional sets, group actions and function spaces, with particular care, since they are concepts that are widely used in the Feit-Thompson theorem proof the team is attacking. Our research results have been published in [46] we co-author with the rest of the Mathematical Components team. In this thesis we also analysed the methodologies we followed and the main issue we encountered in our formalisation experience using the SSReflect Coq extension. Some of these considerations directly influenced some of the work we made as a developer of the Matita interactive theorem prover.

In the second Chapter we describe a formalisation experience we did in the very last part of our PHD, in which we used many of the features we implemented in Matita. We formalised the sandwich theorem, a relevant part of the wider objective of the D.A.M.A. Project, leaded by Claudio Sacerdoti. The project aims to exhibit a constructive proof of the Lebesgue dominated convergence theorem. The sandwich lemma is one of the main components of such proof. This formalisation extensively tests the coercion mechanism we implemented in Matita, building hierarchies of algebraic structures with multiple inheritance. We also describe the methodologies we adopted to overcome some missing features of Matita, like a proper support for setoids in the tactic engine.

The second part of the thesis is dedicated to our experience as a developer of the Matita interactive theorem prover. We worked on many aspects of the system, from its tactic language design and implementation to the integration of searching facilities (Chapter 4), but in this manuscript we concentrate on the refiner (type inference) subsystem and automation.

In Chapter 5 we describe the work we did in the refiner subsystem of Matita to support coercive subtyping and multiple inheritance in an hierarchy of structures. The user is allowed to declare multiple coercive paths from/to the same types. This generates diamonds in the coercion graph, a figure associated with multiple inheritance where common substructures have to be shared. We also describe in details the discipline the user can follow to use fruitfully this technology. This technique has been published in [27] and used to declare the groups and lattices hierarchy for the D.A.M.A. Project described in Chapter 3. When multiple coherent coercive paths are declared, uncommon unification problem arise and we propose a solution that exploits, in the unification algorithm, the information given by the coercion graph. We also implemented support for subset coercions. These coercions are used in the spirit of PVS subtype predicates to specify software written using simple (ML like) types with properties expressed using a richer (dependently) typed language.

The other subsystem we analyse in this dissertation is automation. We developed a tactic that performs rewriting according to the superposition calculus restricted to the unit equality case, tuning it for performances using the huge TPTP [90] library. The obtained results are browsable on the TPTP website, and amount to more than 500 successes on a test suite of 700 problems, of which 350 have been solved in less then ten seconds. Another tactic, performing backward reasoning (Prolog style proof search), has been designed with user interaction in mind. It allows the user to drive the tactic, following or dropping computations. Both tactics are integrated with the searching facilities Matita offers and are able to produce not only CIC proof objects but also proof scripts. The quality of proof scripts is influenced by the proof object the tactic produces, and in the paper [4] we published the proof reconstruction and refinement algorithm used by the former tactic.

During our PHD we put a considerable amount of effort to deliver Matita, reworking the database subsystem to allow an easier installation of the system. We then developed a live CD that allows users to evaluate the system without even installing it on the hard drive. This technology has been successfully used by the students of the Types Summer School 2007. The live CD can also be used without rebooting the computer, since free emulators like VirtualBox[1] and Qemu[2] are able to run the live CD in a virtual machine (actually a regular window on the user's desktop) with a very small slowdown. These emulators are available at no cost for Windows, Mac OS X and Linux based operating systems, the former being straightforward to use. This allows to use MATITA on operating systems, like Windows, to which it has not been ported. Additionally we created a Debian package for MATITA that is easily installable on computers running the Debian GNU/Linux distribution or any of its derivatives like the widespread Ubuntu Linux distribution.

A future direction we intend to follow in the short term is to continue the formalisation of the Lebesgue dominated convergence theorem. That activity will further stress MATITA and its refinement subsystem, contributing to its consolidation.

More in general we believe the MATITA interactive theorem prover has reached a stage in its development that really allows to use it fruitfully to perform formal verification. The system is now pretty easy to install, or run from the live CD, and this is essential to attract users. Thus, seems reasonable to put effort in using the system, possibly formalising relevant results to convince potential users that the system is ready for them.

[1]`http://www.virtualbox.org`

[2]`http://fabrice.bellard.free.fr/qemu/`

Chapter 8

Appendix

8.1 TPTP benchmarks

The following long table reports the result obtained by the auto paramodulation tactic on the problems falling in the unit equality category of the TPTP library. The table reports only the time (in seconds) in case of success, found proofs can be inspected browsing the TPTP website at this url: `http://www.cs.miami.edu/~tptp/cgi-bin/DVTPTP2WWW/view_file.pl?Category=Problems`

Problem	Time	Problem	Time	Problem	Time	Problem	Time
ALG005-1	72.61	ALG006-1	7.07	ALG007-1	8.76	BOO001-1	0.54
BOO002-1	6.47	BOO002-2	9.59	BOO003-2	0.78	BOO003-4	0.48
BOO004-2	0.2	BOO004-4	0.16	BOO005-2	0.32	BOO005-4	0.19
BOO006-2	1.05	BOO006-4	0.7	BOO007-2	31.65	BOO007-4	27.51
BOO008-2	24.61	BOO008-4	146.41	BOO009-2	1.28	BOO009-4	0.26
BOO010-2	0.96	BOO010-4	0.32	BOO011-2	0.08	BOO011-4	0.02
BOO012-2	0.41	BOO012-4	2.11	BOO013-2	0.31	BOO013-4	2.78
BOO014-2	20.42	BOO014-4	114.97	BOO015-2	15.38	BOO015-4	96.29
BOO016-2	0.85	BOO017-2	1.7	BOO018-4	0.02	BOO021-1	0.02
BOO022-1	65.58	BOO023-1	141.4	BOO024-1	17.84	BOO025-1	37.02
BOO026-1	17.55	BOO028-1	FAIL	BOO029-1	17.53	BOO031-1	FAIL
BOO034-1	0.89	BOO067-1	FAIL	BOO068-1	2.99	BOO069-1	2.85
BOO070-1	2.76	BOO071-1	3.18	BOO072-1	62.23	BOO073-1	FAIL
BOO074-1	99.57	BOO075-1	4.04	BOO076-1	FAIL	COL001-1	356.14
COL001-2	4.21	COL002-1	4.18	COL002-4	FAIL	COL002-5	FAIL
COL003-1	FAIL	COL004-1	FAIL	COL004-3	0.02	COL006-1	FAIL
COL006-5	FAIL	COL006-6	FAIL	COL006-7	FAIL	COL007-1	0
COL008-1	0.01	COL009-1	3.16	COL010-1	1.59	COL011-1	FAIL
COL012-1	0	COL013-1	0.01	COL014-1	0	COL015-1	0.02
COL016-1	0.01	COL017-1	0.01	COL018-1	0	COL019-1	1.65
COL020-1	FAIL	COL021-1	0.03	COL022-1	0.05	COL023-1	FAIL

Problem	Time	Problem	Time	Problem	Time	Problem	Time
COL024-1	0.01	COL025-1	0.05	COL026-1	FAIL	COL027-1	FAIL
COL029-1	0	COL030-1	0.18	COL031-1	0.01	COL032-1	0.05
COL033-1	4.71	COL034-1	0.51	COL035-1	FAIL	COL036-1	FAIL
COL037-1	FAIL	COL038-1	FAIL	COL039-1	5.87	COL041-1	0.65
COL042-1	FAIL	COL042-6	FAIL	COL042-7	FAIL	COL042-8	FAIL
COL042-9	FAIL	COL043-1	FAIL	COL043-3	FAIL	COL044-1	FAIL
COL044-6	FAIL	COL044-7	FAIL	COL044-8	FAIL	COL044-9	FAIL
COL045-1	0.22	COL046-1	FAIL	COL048-1	0.06	COL049-1	5.11
COL050-1	0.01	COL051-1	0.01	COL052-1	FAIL	COL053-1	0.01
COL056-1	0.04	COL057-1	FAIL	COL058-1	0.32	COL058-2	0.02
COL058-3	0.01	COL059-1	FAIL	COL060-1	23.48	COL060-2	0
COL060-3	0	COL061-1	477.77	COL061-2	0	COL061-3	0
COL062-1	FAIL	COL062-2	0	COL062-3	0	COL063-1	FAIL
COL063-2	0	COL063-3	0	COL063-4	0	COL063-5	0
COL063-6	0	COL064-1	FAIL	COL064-2	0	COL064-3	0
COL064-4	0	COL064-5	0	COL064-6	0	COL064-7	0
COL064-8	0	COL064-9	0	COL065-1	FAIL	COL066-1	FAIL
COL066-2	2.51	COL066-3	2.07	COL070-1	FAIL	COL075-2	0.16
COL083-1	0	COL084-1	0	COL085-1	0	COL086-1	0
GRP001-2	0.03	GRP001-4	0.18	GRP002-2	31.68	GRP002-3	347.29
GRP002-4	67.57	GRP010-4	0.05	GRP011-4	0.25	GRP012-4	0.04
GRP014-1	50.29	GRP022-2	0.01	GRP023-2	0.01	GRP024-5	FAIL
GRP114-1	215.54	GRP115-1	0.05	GRP116-1	0.15	GRP117-1	0.06
GRP118-1	0.17	GRP119-1	54.79	GRP120-1	41.58	GRP121-1	37.22
GRP122-1	38.77	GRP136-1	0.1	GRP137-1	0.1	GRP138-1	4.32
GRP139-1	0.09	GRP140-1	3.11	GRP141-1	0.21	GRP142-1	0.04
GRP143-1	0.05	GRP144-1	0.04	GRP145-1	0.05	GRP146-1	0.09
GRP147-1	4.68	GRP148-1	3.4	GRP149-1	0.24	GRP150-1	0.04

Problem	Time	Problem	Time	Problem	Time	Problem	Time
GRP151-1	0.06	GRP152-1	0.04	GRP153-1	0.05	GRP154-1	0.1
GRP155-1	0.08	GRP156-1	0.13	GRP157-1	0.08	GRP158-1	0.13
GRP159-1	0.28	GRP160-1	0.05	GRP161-1	0.03	GRP162-1	0.29
GRP163-1	0.43	GRP164-1	FAIL	GRP164-2	FAIL	GRP165-1	54.54
GRP165-2	51.75	GRP166-1	261.59	GRP166-2	FAIL	GRP166-3	62.82
GRP166-4	52.29	GRP167-1	124.19	GRP167-2	300.19	GRP167-3	311.96
GRP167-4	250.37	GRP167-5	19.53	GRP168-1	0.09	GRP168-2	0.09
GRP169-1	107.13	GRP169-2	114.44	GRP170-1	307.4	GRP170-2	318.68
GRP170-3	339.79	GRP170-4	325.63	GRP171-1	6.36	GRP171-2	6.96
GRP172-1	10.72	GRP172-2	6.19	GRP173-1	15.46	GRP174-1	21.49
GRP175-1	33.68	GRP175-2	41.46	GRP175-3	41.14	GRP175-4	41.87
GRP176-1	0.11	GRP176-2	0.08	GRP177-2	FAIL	GRP178-1	282.09
GRP178-2	297.82	GRP179-1	FAIL	GRP179-2	FAIL	GRP179-3	FAIL
GRP180-1	FAIL	GRP180-2	FAIL	GRP181-1	FAIL	GRP181-2	FAIL
GRP181-3	547.55	GRP181-4	FAIL	GRP182-1	0.06	GRP182-2	0.05
GRP182-3	0.04	GRP182-4	0.05	GRP183-1	FAIL	GRP183-2	FAIL
GRP183-3	FAIL	GRP183-4	FAIL	GRP184-1	FAIL	GRP184-2	FAIL
GRP184-3	FAIL	GRP184-4	39.14	GRP185-1	FAIL	GRP185-2	FAIL
GRP185-3	FAIL	GRP185-4	FAIL	GRP186-1	FAIL	GRP186-2	FAIL
GRP186-3	0.21	GRP186-4	0.13	GRP187-1	FAIL	GRP188-1	0.03
GRP188-2	0.05	GRP189-1	0.06	GRP189-2	0.07	GRP190-1	88.61
GRP190-2	81.56	GRP191-1	106.88	GRP191-2	89.87	GRP192-1	0.69
GRP193-1	70.37	GRP193-2	50.43	GRP195-1	FAIL	GRP196-1	FAIL
GRP200-1	FAIL	GRP201-1	FAIL	GRP202-1	FAIL	GRP203-1	FAIL
GRP205-1	FAIL	GRP206-1	0.18	GRP403-1	8.97	GRP404-1	51.68
GRP405-1	33.55	GRP406-1	128.4	GRP407-1	240.49	GRP408-1	243.34
GRP409-1	10.16	GRP410-1	22.28	GRP411-1	19.42	GRP412-1	19.16
GRP413-1	100	GRP414-1	62.5	GRP415-1	342.07	GRP416-1	400.72

Problem	Time	Problem	Time	Problem	Time	Problem	Time
GRP417-1	388.08	GRP418-1	FAIL	GRP419-1	FAIL	GRP420-1	FAIL
GRP421-1	3.56	GRP422-1	150.83	GRP423-1	76.62	GRP424-1	11.61
GRP425-1	11.2	GRP426-1	20.01	GRP427-1	31.94	GRP428-1	36.6
GRP429-1	50.34	GRP430-1	17.74	GRP431-1	16.07	GRP432-1	15.53
GRP433-1	4.91	GRP434-1	5.55	GRP435-1	8.83	GRP436-1	36.79
GRP437-1	35.27	GRP438-1	38.97	GRP439-1	13.91	GRP440-1	28.28
GRP441-1	36.2	GRP442-1	163.86	GRP443-1	169.09	GRP444-1	139.87
GRP445-1	0.48	GRP446-1	0.1	GRP447-1	1.11	GRP448-1	0.33
GRP449-1	0.1	GRP450-1	1.05	GRP451-1	0.15	GRP452-1	1.64
GRP453-1	2.88	GRP454-1	0	GRP455-1	0.05	GRP456-1	0.24
GRP457-1	0	GRP458-1	0.06	GRP459-1	0.25	GRP460-1	0
GRP461-1	0.02	GRP462-1	0.31	GRP463-1	0	GRP464-1	0.02
GRP465-1	0.4	GRP466-1	0.1	GRP467-1	0.67	GRP468-1	1.07
GRP469-1	116.49	GRP470-1	123.5	GRP471-1	48.37	GRP472-1	41.32
GRP473-1	37.76	GRP474-1	53.59	GRP475-1	51.86	GRP476-1	37.8
GRP477-1	41.41	GRP478-1	18.96	GRP479-1	19.22	GRP480-1	21.92
GRP481-1	0.03	GRP482-1	0.05	GRP483-1	1.16	GRP484-1	0.11
GRP485-1	0.23	GRP486-1	1.23	GRP487-1	0.04	GRP488-1	1.17
GRP489-1	2.53	GRP490-1	0.06	GRP491-1	0.08	GRP492-1	0.36
GRP493-1	0.03	GRP494-1	0.06	GRP495-1	0.27	GRP496-1	0.01
GRP497-1	0.19	GRP498-1	0.46	GRP499-1	7.75	GRP500-1	4.26
GRP501-1	87.52	GRP502-1	161.64	GRP503-1	248.32	GRP504-1	247.37
GRP505-1	FAIL	GRP506-1	FAIL	GRP507-1	FAIL	GRP508-1	FAIL
GRP509-1	143.03	GRP510-1	0.78	GRP511-1	8.71	GRP512-1	0.17
GRP513-1	57.79	GRP514-1	0.17	GRP515-1	0.31	GRP516-1	0.17
GRP517-1	1.38	GRP518-1	0.13	GRP519-1	0.46	GRP520-1	0.18
GRP521-1	0.12	GRP522-1	0.09	GRP523-1	2.7	GRP524-1	0.19
GRP525-1	0.14	GRP526-1	0.04	GRP527-1	9.44	GRP528-1	0.19

Problem	Time	Problem	Time	Problem	Time	Problem	Time
GRP529-1	0.15	GRP530-1	0.06	GRP531-1	2.91	GRP532-1	0.15
GRP533-1	0.01	GRP534-1	0.03	GRP535-1	2.82	GRP536-1	0.09
GRP537-1	0.01	GRP538-1	0.02	GRP539-1	2.45	GRP540-1	0.13
GRP541-1	0	GRP542-1	0.02	GRP543-1	0.67	GRP544-1	0.05
GRP545-1	0	GRP546-1	0.01	GRP547-1	1.7	GRP548-1	0.06
GRP549-1	0	GRP550-1	0.02	GRP551-1	0.75	GRP552-1	0.03
GRP553-1	0.13	GRP554-1	0.11	GRP555-1	1.03	GRP556-1	0.09
GRP557-1	0.16	GRP558-1	0.1	GRP559-1	3.08	GRP560-1	0.25
GRP561-1	0.14	GRP562-1	0.07	GRP563-1	6.03	GRP564-1	0.16
GRP565-1	0.09	GRP566-1	0.06	GRP567-1	28.2	GRP568-1	0.36
GRP569-1	0.07	GRP570-1	0.06	GRP571-1	26.98	GRP572-1	0.24
GRP573-1	0.06	GRP574-1	0.1	GRP575-1	46.52	GRP576-1	0.35
GRP577-1	0.21	GRP578-1	0.43	GRP579-1	1.66	GRP580-1	0.46
GRP581-1	0.04	GRP582-1	0.3	GRP583-1	2.94	GRP584-1	0.23
GRP585-1	0.11	GRP586-1	0.09	GRP587-1	FAIL	GRP588-1	1.47
GRP589-1	0.5	GRP590-1	0.12	GRP591-1	0.64	GRP592-1	0.28
GRP593-1	2.56	GRP594-1	1.16	GRP595-1	0.24	GRP596-1	0.29
GRP597-1	0.62	GRP598-1	0.28	GRP599-1	2.33	GRP600-1	0.36
GRP601-1	3.56	GRP602-1	0.44	GRP603-1	0.56	GRP604-1	0.77
GRP605-1	1.75	GRP606-1	1.62	GRP607-1	FAIL	GRP608-1	2.11
GRP609-1	2.03	GRP610-1	0.5	GRP611-1	0.24	GRP612-1	0.32
GRP613-1	0.78	GRP614-1	0.3	GRP615-1	1.73	GRP616-1	0.32
LAT006-1	38.21	LAT007-1	97.83	LAT008-1	0.82	LAT009-1	341.28
LAT010-1	FAIL	LAT011-1	FAIL	LAT012-1	4.82	LAT013-1	90.79
LAT014-1	0	LAT017-1	FAIL	LAT018-1	FAIL	LAT019-1	FAIL
LAT020-1	FAIL	LAT021-1	FAIL	LAT022-1	FAIL	LAT023-1	FAIL
LAT026-1	20.28	LAT027-1	13.64	LAT028-1	17.73	LAT031-1	0.99
LAT032-1	27.63	LAT033-1	0	LAT034-1	0	LAT038-1	FAIL

Problem	Time	Problem	Time	Problem	Time	Problem	Time
LAT039-1	0.07	LAT039-2	0.05	LAT040-1	36.21	LAT042-1	1.93
LAT043-1	158.67	LAT044-1	FAIL	LAT045-1	0.17	LAT070-1	FAIL
LAT072-1	FAIL	LAT074-1	FAIL	LAT075-1	FAIL	LAT076-1	FAIL
LAT077-1	FAIL	LAT078-1	FAIL	LAT079-1	FAIL	LAT080-1	84.86
LAT081-1	FAIL	LAT082-1	FAIL	LAT083-1	89	LAT084-1	FAIL
LAT085-1	FAIL	LAT086-1	573.47	LAT087-1	588.84	LAT088-1	0.06
LAT089-1	1.68	LAT090-1	0.14	LAT091-1	16.44	LAT092-1	82.68
LAT093-1	320.09	LAT094-1	97.53	LAT095-1	474.76	LAT096-1	162.98
LAT097-1	115.99	LAT138-1	FAIL	LAT139-1	FAIL	LAT140-1	FAIL
LAT141-1	FAIL	LAT142-1	FAIL	LAT143-1	FAIL	LAT144-1	FAIL
LAT145-1	FAIL	LAT146-1	FAIL	LAT147-1	FAIL	LAT148-1	FAIL
LAT149-1	FAIL	LAT150-1	FAIL	LAT151-1	FAIL	LAT152-1	FAIL
LAT153-1	FAIL	LAT154-1	FAIL	LAT155-1	FAIL	LAT156-1	FAIL
LAT157-1	FAIL	LAT158-1	FAIL	LAT159-1	FAIL	LAT160-1	FAIL
LAT161-1	FAIL	LAT162-1	FAIL	LAT163-1	FAIL	LAT164-1	FAIL
LAT165-1	FAIL	LAT166-1	FAIL	LAT167-1	FAIL	LAT168-1	190.3
LAT169-1	FAIL	LAT170-1	FAIL	LAT171-1	250.07	LAT172-1	FAIL
LAT173-1	FAIL	LAT174-1	FAIL	LAT175-1	FAIL	LAT176-1	FAIL
LAT177-1	FAIL	LCL109-2	FAIL	LCL109-6	FAIL	LCL110-2	1.67
LCL111-2	5.31	LCL112-2	1.81	LCL113-2	1.39	LCL114-2	1.9
LCL115-2	1.41	LCL116-2	5.72	LCL132-1	0.02	LCL133-1	0.04
LCL134-1	0.05	LCL135-1	0.05	LCL138-1	FAIL	LCL139-1	1.81
LCL140-1	1.45	LCL141-1	2.09	LCL153-1	1.36	LCL154-1	10.65
LCL155-1	0.14	LCL156-1	1.38	LCL157-1	1.13	LCL158-1	1.87
LCL159-1	131.67	LCL160-1	FAIL	LCL161-1	0.1	LCL162-1	FAIL
LCL163-1	10.1	LCL164-1	0.15	LDA001-1	0.13	LDA002-1	22.3
LDA007-3	0.08	RNG007-4	0.06	RNG008-3	17.75	RNG008-4	18.52
RNG008-7	47.63	RNG009-5	FAIL	RNG009-7	FAIL	RNG011-5	0.14

Problem	Time	Problem	Time	Problem	Time	Problem	Time
RNG012-6	114.29	RNG013-6	132.92	RNG014-6	145.08	RNG015-6	127.35
RNG016-6	133.38	RNG017-6	120.96	RNG018-6	116.44	RNG019-6	FAIL
RNG019-7	FAIL	RNG020-6	FAIL	RNG020-7	FAIL	RNG021-6	FAIL
RNG021-7	FAIL	RNG023-6	0.09	RNG023-7	0.11	RNG024-6	0.09
RNG024-7	0.11	RNG025-4	FAIL	RNG025-5	FAIL	RNG025-6	FAIL
RNG025-7	FAIL	RNG026-6	FAIL	RNG026-7	FAIL	RNG027-5	FAIL
RNG027-7	FAIL	RNG027-8	FAIL	RNG027-9	FAIL	RNG028-5	FAIL
RNG028-7	FAIL	RNG028-8	FAIL	RNG028-9	FAIL	RNG029-5	FAIL
RNG029-6	FAIL	RNG029-7	FAIL	RNG035-7	FAIL	ROB001-1	FAIL
ROB002-1	0.1	ROB003-1	0.72	ROB004-1	66.26	ROB005-1	FAIL
ROB006-1	FAIL	ROB006-2	FAIL	ROB008-1	80.62	ROB009-1	0.07
ROB010-1	0.03	ROB013-1	0.02	ROB022-1	FAIL	ROB023-1	FAIL
ROB026-1	FAIL	ROB030-1	0.44	ROB031-1	FAIL	ROB032-1	FAIL
SYN080-1	0	SYN083-1	0				

8.2 Tinycals utility functions

The goal automatically selected by "[" or "|" is called *unhandled* until a tactic is applied to it. Unhandled goals are just postponed (not moved into the todo list τ) by $i_1, \ldots, i_n$":". Goals opened by a tactic are marked with *mark_as_handled* to distinguishing them from unhandled goals. Goals marked with `Closed` are always considered unhandled. The function *renumber_branches* is used by "[" to name branches.

$$\mathit{unhandled}(l) = \begin{cases} \mathit{true} & \text{if} \quad l = \langle n, \texttt{Open } g\rangle \wedge n > 0 \quad \vee \quad l = \langle \_, \texttt{Closed } g\rangle \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{mark\_as\_handled}([g_1; \cdots; g_n]) = [\langle 0, \texttt{Open } g_1\rangle; \cdots; \langle 0, \texttt{Open } g_n\rangle]$$

$$\mathit{renumber\_branches}([\langle i_1, s_1\rangle; \cdots; \langle i_n, s_n\rangle]) = [\langle 1, s_1\rangle; \cdots; \langle n, s_n\rangle]$$

The next three functions returns open goals or tasks in the status or parts of it. Open goals are those corresponding to conjectures still to be proved.

$$\mathit{get\_open\_tasks}(l) =$$
$$\begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Open } g\rangle :: \mathit{get\_open\_tasks}(\mathit{tl}) & \text{if } l = \langle i, \texttt{Open } g\rangle :: \mathit{tl} \\ \mathit{get\_open\_tasks}(\mathit{tl}) & \text{if } l = \mathit{hd} :: \mathit{tl} \end{cases}$$

$$\mathit{get\_open\_goals\_in\_tasks\_list}(l) =$$
$$\begin{cases} [\,] & \text{if } l = [\,] \\ g :: \mathit{get\_open\_goals\_in\_tasks\_list}(\mathit{tl}) & \text{if } l = \langle \_, \texttt{Open } g\rangle :: \mathit{tl} \\ \mathit{get\_open\_goals\_in\_tasks\_list}(\mathit{tl}) & \text{if } l = \langle \_, \texttt{Closed } g\rangle :: \mathit{tl} \end{cases}$$

$$\mathit{get\_open\_goals\_in\_status}(S) =$$
$$\begin{cases} [\,] & \text{if } S = [\,] \\ \mathit{get\_open\_goals\_in\_tasks\_list}(\Gamma @ \tau @ \kappa) & \\ \quad @\mathit{get\_open\_goals\_in\_status}(\mathit{tl}) & \text{if } S = \langle \Gamma, \tau, \kappa, \_\rangle :: \mathit{tl} \end{cases}$$

To keep the correspondence between branches in the script and ramifications in the proof, goals closed by side-effects are marked as `Closed` if they are in Γ (that keeps track of open branches). Otherwise they are silently removed from postponed goals (in todo list τ or dot continuation κ). `Closed` branches have to be accepted by the user with “`skip`”.

$$\mathit{close\_tasks}(G, S) =$$
$$\begin{cases} [\,] & \text{if } S = [\,] \\ \langle \mathit{close}_{\mathit{aux}}(G, \Gamma), \tau', \kappa', t\rangle :: \mathit{close\_tasks}(G, \mathit{tl}) & \text{if } S = \langle \Gamma, \tau, \kappa, t\rangle :: \mathit{tl} \\ \quad \text{where } \tau' = \mathit{remove\_tasks}(G, \tau) & \\ \quad \text{and } \kappa' = \mathit{remove\_tasks}(G, \kappa) & \end{cases}$$

$$\mathit{close}_{\mathit{aux}}(G, l) =$$
$$\begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Closed } g\rangle :: \mathit{close}_{\mathit{aux}}(G, \mathit{tl}) & \text{if } l = \langle i, \texttt{Open } g\rangle :: \mathit{tl} \wedge g \in G \\ \mathit{hd} :: \mathit{close}_{\mathit{aux}}(G, \mathit{tl}) & \text{if } l = \mathit{hd} :: \mathit{tl} \end{cases}$$

$$
\mathit{remove\_tasks}(G, l) =
$$

$$
\begin{cases}
[\,] & \text{if } l = [\,] \\
\mathit{remove\_tasks}(G, \mathit{tl}) & \text{if } l = \langle i, \texttt{Open}\ g \rangle :: \mathit{tl} \wedge g \in G \\
\mathit{hd} :: \mathit{remove\_tasks}(G, \mathit{tl}) & \text{if } l = \mathit{hd} :: \mathit{tl}
\end{cases}
$$

References

[1] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Post-proceedings of the Types 2004 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.

[2] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Proceedings of Types 2006: Conference of the Types Project. Nottingham, UK – April 18-21*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interfaces for Theorem Proving. To appear.

[4] Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In *Calculemus/MKM*, pages 146–160, 2007.

[5] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2000.

[6] David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1–2):273–309, 2001.

[7] Anthony Bailey. Coercion synthesis in computer implementations of type-theoretic frameworks. In *TYPES '96: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 9–27, London, UK, 1998. Springer-Verlag.

[8] Anthony Bailey. *The Machine-Checked Literate Formalisation Of Algebra In Type Theory*. PhD thesis, University of Manchester, 1998.

[9] Marian Alexandru Baroni. *The constructive theory of Riesz spaces and applications in mathematical economics*. PhD thesis, University of Canterbury, department of mathematics and statistics, 2004.

[10] G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, March 2003.

[11] Gilles Barthe. Implicit coercions in type systems. In *Types for Proofs and Programs: International Workshop, TYPES 1995*, pages 1–15, 1995.

[12] Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.

[13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer Verlag, 2004. ISBN-3-540-20854-2.

[14] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Software (STACS)*, volume 789 of *Lecture Notes in Computer Science*, 1994.

[15] Gustavo Betarte and Alvaro Tasistro. Formalization of systems of algebras using dependent record types and subtyping: An example. In *Proceedings of the 7th. Nordic workshop on Programming Theory, Gothenburg*, 1995.

[16] Gustavo Betarte and Alvaro Tasistro. Extension of martin-löf's type theory

with record types and subtyping. In *Twenty-five Years of Constructive Type Theory.* Oxford Science Publications, 1998.

[17] Bridges D. Bishop E. *Constructive Analysis.* Springer Verlag, 1985.

[18] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 515–529. Springer-Verlag, 1997.

[19] Arnim Buch, Thomas Hillenbrand, and Roland Fettig. WALDMEISTER: High performance equational theorem proving. In *Design and Implementation of Symbolic Computation Systems*, pages 63–64, 1996.

[20] P. Callaghan. Coherence checking of coercions in plastic. In *In Proc. Workshop on Subtyping and Dependent Types in Programming*, 2000.

[21] Paul Callaghan and Zhaohui Luo. Implementation techniques for inductive types in plastic. In *TYPES*, pages 94–113, 1999.

[22] Gang Chen. *Subtyping, Type Conversion and Transitivity Elimination.* PhD thesis, University Paris 7, 1998.

[23] Gang Chen. Coercive subtyping for the calculus of constructions. In *The 30th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 2003.

[24] Claudio Sacerdoti Coen. Mathematical libraries as proof assistant environments. In *MKM*, pages 332–346, 2004.

[25] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In *TYPES*, pages 98–114, 2004.

[26] Claudio Sacerdoti Coen. A declarative language for Matita. In *PLMMS, Workshop on Programming Languages for Mechanized Mathematics*, 2007. To appear.

[27] Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *TYPES*, 2007. To appear.

[28] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[29] The Coq proof-assistant.
`http://coq.inria.fr`.

[30] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[31] Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. *Fundamenta Informaticae*, 65(1-2):113–134, 2005.

[32] Yann Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.

[33] Yann Coscoy, Gilles Kahn, and Laurent Thery. Extracting Text from Proofs. Technical Report RR-2459, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1995.

[34] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *MKM*, pages 88–103, 2004.

[35] W. Mc Cune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[36] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95. Springer-Verlag, November 2000.

[37] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[38] Fremlin D.H. *Topological Riesz Spaces and Measure Theory.* Cambridge University Press, 1974.

[39] Bruno Dutertre and Leonardo de Moura. The yices smt solver.

[40] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), 2000.

[41] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3):775–1029, 1963.

[42] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *TYPES*, pages 78–92, 1998.

[43] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.

[44] Georges Gonthier. A computer-checked proof of the four-colour theorem. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[45] Georges Gonthier. Notations of the four colour thorem proof. Available at `http://research.microsoft.com/~gonthier/4colnotations.pdf`.

[46] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Thery. A modular formalisation of finite group theory. In *The 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732, pages 86–101, 2007.

[47] B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml.* Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France, December 2003.

[48] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *TPHOLs*, pages 98–113, 2005.

[49] Ferruccio Guidi. Procedural representation of cic proof terms. In *PLMMS, Workshop on Programming Languages for Mechanized Mathematics*, 2007. To appear.

[50] Weber H. *Uniform lattices I. A generalization of topological Riesz spaces and topological Boolean rings.* Ann. Mat. Pure Appl., 1991.

[51] Weber H. *Uniform lattices II. Order continuity and exhaustivity.* Ann. Mat. Pure Appl., 1993.

[52] John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer-Verlag, 1996.

[53] Michael Hedberg. Unpublished proof formalized in lego by T. Kleymann and in coq by B. Barras. `http://coq.inria.fr/library/Coq.Logic.Eqdep_dec.html`.

[54] The Isabelle proof-assistant. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[55] Alex P. Jones, Zhaohui Luo, and Sergei Soloviev. Some algorithmic and proof-theoretical aspects of coercive subtyping. In *TYPES*, pages 173–195, 1996.

[56] B. Löchner and Th. Hillenbrand. A phytography of WALDMEISTER. *AI Communications*, 15(2–3):127–133, 2002.

[57] Zhaohui Luo. Coercive subtyping in type theory. In *CSL*, pages 276–296, 1996.

[58] Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.

[59] Zhaohui Luo and Sergei Soloviev. Dependent coercions. *Electr. Notes Theor. Comput. Sci.*, 29, 1999.

[60] E. Lusk, W. McCune, and R. Overbeek. Logic machine architecture: Kernel functions. In editor D. Loveland, editor, *Proceedings of the 6th Conference on Automated Deduction*, volume 138, pages 70–84, 1982.

[61] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The krakatoa tool for certificationof java/javacard programs annotated in jml. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.

[62] Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.

[63] William McCune and Larry Wos. Otter - the cade-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[64] The Mizar proof-assistant.
`http://mizar.uwb.edu.pl/`.

[65] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory.* PhD thesis, INRIA, November 1997.

[66] Sara Negri. Sequent calculus proof theory of intuitionistic apartness and order relations. *Archive for Mathematical Logic*, 38/ 8:521–547, 1999.

[67] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. *Lecture Notes in Computer Science*, 2083:257–??, 2001.

[68] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. In *IJCAR*, pages 257–271, 2001.

[69] Robert Nieuwenhuis and Alberto Rubio. *Paramodulation-based thorem proving.* Elsevier and MIT Press, 2001. ISBN-0-262-18223-8.

[70] OMDoc: An open markup format for mathematical documents (draft, version 1.2).
`http://www.mathweb.org/omdoc/pubs/omdoc1.2.pdf`, 2005.

[71] Luca Padovani. *MathML Formatting*. PhD thesis, University of Bologna, February 2003. Technical Report UBLCS 2003-03.

[72] Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In *Proceedings of Mathematical Knowledge Management 2006*, volume 4108 of *Lectures Notes in Artificial Intelligence*, pages 194–207. Springer-Verlag, 2006.

[73] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supŕieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[74] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[75] The PVS specification and verification system.
`http://pvs.csl.sri.com/`.

[76] Alexandre Riazanov. *Implementing an Efficient Theorem Prover*. PhD thesis, The University of Manchester, 2003.

[77] Alexandre Riazanov and Andrei Voronkov. Partially adaptive code trees. In *JELIA*, pages 209–223, 2000.

[78] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.

[79] Walter Rudin. *Principles of Mathematical Analysis (International Series in Pure & Applied Mathematics)*. McGraw-Hill Publishing Co., September 1976.

[80] Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.

[81] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.

[82] Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.

[83] Claudio Sacerdoti Coen and Stefano Zacchiroli. Spurious disambiguation error detection. In *Proceedings of Mathematical Knowledge Management 2007*, volume 4573 of *Lecture Notes in Artificial Intelligence*, pages 381–392. Springer-Verlag, 2007.

[84] Amokrane Saibi. Typing algorithm in type theory with inheritance. In *The 24th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 1997.

[85] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in pvs. In *WADT*, pages 37–52, 1999.

[86] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. In *Computers and Mathematics with Applications*, 1993.

[87] Matthieu Sozeau. Subset coercions in coq. In *TYPES*, pages 237–252, 2006.

[88] B Spitters. Constructive algebraic integration theory. *Ann. Pure Appl. Logic*, 137:380–390, 2006.

[89] Martin Strecker. *Construction and Deduction in Type Theories.* PhD thesis, Universität Ulm, 1998.

[90] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[91] Enrico Tassi. Studio e implementazione della gerarchia degli universi in sistemi di supporto al ragionamento formale basati sulla teoria dei tipi. Master's thesis, University of Bologna, 2004.

[92] The Coq Development Team. The Coq proof assistant reference manual. `http://coq.inria.fr/doc/main.html`, 2005.

[93] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, May 1994.

[94] Freek Wiedijk. Mmode, a mizar mode for the proof assistant coq. Technical Report NIII-R0333, University of Nijmegen, 2003.

[95] Stefano Zacchiroli. Web services per il supporto alla dimostrazione interattiva. Master's thesis, University of Bologna, 2003.

[96] Stefano Zacchiroli. *User Interaction Widgets for Interactive Theorem Proving.* PhD thesis, University of Bologna, 2007.

[97] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.