Master thesis: Equality reasoning in MMT

Sven Wille

09.05.2022

Contents

Abstract

In recent developments MMT got extended with an interactive proof system which contained only basic tactics. This did not include equality reasoning. To turn MMT into something of a useful prover one has to add capabilities for equality reasoning, since doing rewriting with only low level application of transitivity and congruence theorems is fairly difficult. This work presents the reader with an extension to the current way MMT does equality reasoning (interactive or not). This was achieved by extending the interactive prover and also by adding new constructs to plain MMT. The results include new tactics for rewriting, a solver for reasoning about algebraic structures and a special construct for displaying chained equality/transitivity.

1 Introduction

Motivation Formal systems allow the user to formalize, make statements and make proofs about mathematical systems. Without the help of computer guided systems this becomes quite difficult and prone to errors. MMT [17], as such a system, aims, among other things, at enabling users to formalize and verify mathematical systems. Yet, rather unintuitively, up until recently it offered only limited support for (interactive) proof development. Proofs had to be given as a completely finished proof in form of a lambda term rather than being build incrementally using something that abstracts from the lambda term structure. In a rather new development, mainly carried out by the author of this work, MMT was extended with some limited capabilities for interactive proof development.
After MMT was extended with the capabilities mentioned above it became clear rather quickly that one of most important features missing from MMT in general and from the interactive prover in particular is varied support for reasoning about equalities, like rewriting. Without these kinds of extensions a vast majority of proofs become borderline unbearable to make due to being to tedious to construct since the user has to manually keep track of a lot of things.

Contribution This work's main focus lies on extending MMT with support for equality reasoning for the interactive prover front end as well as plain

MMT. For the interactive system, rewriting and automated algebraic reasoning were added. For the non-interactive system, some constructs where introduced that allow for easy proof construction of transitive (equality) chains. The results presented here are, just like the authors previous work [31], guided by already existing systems, mainly Agda, Coq and Isabelle(HOL). These three systems works fairly different when it comes to equality reasoning. Isabelle [9] focuses on automated reasoning while Coq is using mostly its tactics [23] based system to work on equalities. Agda, is a system that tries not to abstract to much from the actual proof (i.e. lambda terms) offers special support to work on equalities while still being fairly concrete. When it comes to automation though, all three "role model systems" converge towards offering push button solutions that all work fairly similarly.

Overview Chapter two gives a short overview of MMT, LF, some of the inner workings of MMT and the proof of concept interactive theorem prover that was developed for MMT in the authors previous work. Chapter three gives an overview of the state of the art. Chapter four presents a new way of writing proofs that form a transitivity chain. Chapter five presents extensions to the interactive prover for rewriting. Chapter six introduces some slight automated reasoning for the interactive prover concerning algebraic structures. Chapter seven is conclusion and future work. Appendix A is a very brief tutorial, meant to be a kick-start for new users to the interactive proof system. Appendix B is an index of all newly implemented tactics that are part of the interactive proof language.

2 Preliminaries

This section gives a short overview of MMT, LF. It is by no means complete and will only focus on what is needed to understand the rest of this work. For example integral parts of MMT like structures are left out.

2.1 MMT

MMT is a framework for implementing different formal systems. It abstracts from theoretical and practical aspects of type theoretical and logical foundations of these systems [16]. MMT itself is build on a small set of carefully chosen, orthogonal primitives. It makes a distinction between large scale

THEORY	::=	theory NAME [: METATHEORY] = (THEORY*|INCLUDE*|CONSTANT*)
NAME	::=	string
URI	::=	uri-string
METATHEORY	::=	URI
INCLUDE	::=	include URI
CONSTANT	::=	NAME[= MMT-OBJ][: MMT-OBJ]

Figure 1: MMT grammar

concerns and small scale concerns. Large scale concerns are dealt with on the MMT level (i.e. generically) and small scale concerns are dealt with by individual foundational languages. MMT goes beyond the meta-logical-framework approach in that it does not commit to a particular meta logic, instead allowing for different foundations.
Because the MMT-level already provides common features, implementing new foundations requires less work. Instead of starting from scratch, every new foundation only needs to add necessary extensions, mainly in the form of so-called "rules", and notations to MMT. A rule itself is, plainly spoken, just some Scala code which gets loaded dynamically during run-time. Notations on the other hand are declared in MMT source files (not to be confused with the source of MMT).

MMT-syntax MMT consists of theories which in turn are lists of constant declarations [17]. Each constant declaration itself can be made up of a definition, a type declaration, and several optional notations.
There are some exceptions to that like import declarations, which only consist of the import path. A theory can have a meta theory which is referenced as a valid uri-string (for example "ur:?LF" or in long form "http://cds.omdoc.org/urtheories:?LF").
A semi-formal definition of the syntax is defined in figure 1.

MMT-objects "MMT-object" is a general name for most elements in an MMT-file. MMT-objects are build of simple building blocks as described in an oversimplified version in figure 2. To not loose focus, only the parts needed for the definition and type of a constant as described in figure 1 are included.
Each basic building block has a special purpose :

MMT-OBJ	::=	OMV|OML|OMA|OMID|OMBINDC
VARDECL	::=	VARDECL(LOCALNAME, [MMT-OBJ] , [MMT-OBJ])
LOCALNAME	::=	string
OMV	::=	OMV(string)
OMID	::=	OMID(uri-string)
OML	::=	OML(LOCALNAME , [MMT-OBJ] , [MMT-OBJ])
OMA	::=	OMA(MMT-OBJ, MMT-OBJ*)
OMBINDC	::=	OMBINDC(MMT-OBJ, VARDECL* , MMT-OBJ)

Figure 2: MMT-object grammar

- OMV : (local) variable name
- OMID: reference/name of a constant in the (global) context
- OML : local declarations with an optional type (second argument) and optional definition (third argument)
- OMA : application of terms (arguments) to a term
- OMBINDC : binder block, used for things like the argument binder in a lambda term
- LOCALNAME : a local identifier (for simplicity it is defined as an alias for string)
- VARDECL : like OML but grammatically separated from OML (for reasons that go beyond this work)

Checking in MMT When MMT checks an MMT-file it goes through three phases as depicted in figure 3. During parsing MMT checks whether a term is syntactically valid. Beyond that the MMT-parser also adds dependencies between variables and unknowns(i.e. which variable the solution for the term that the unknown represents can depend on). The way dependencies are added follows the structure of lambda terms. For example in the term “[x] ([y,f] a) ([z] b)” the variable “a” can depend on x, y and f but not on “z”.
The type checking phase checks that terms a valid with regard to typing. This phase can create constraints that are tried to be proven during the third, “proving”, phase.

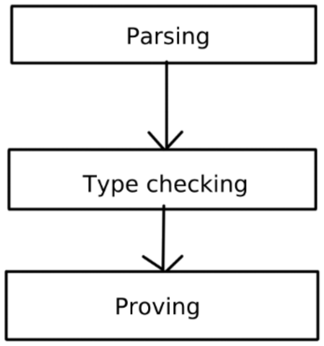
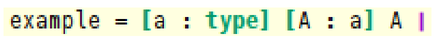
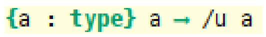


Figure 3: different phases MMT goes through when checking a file



```
example = [a : type] [A : a] A
```

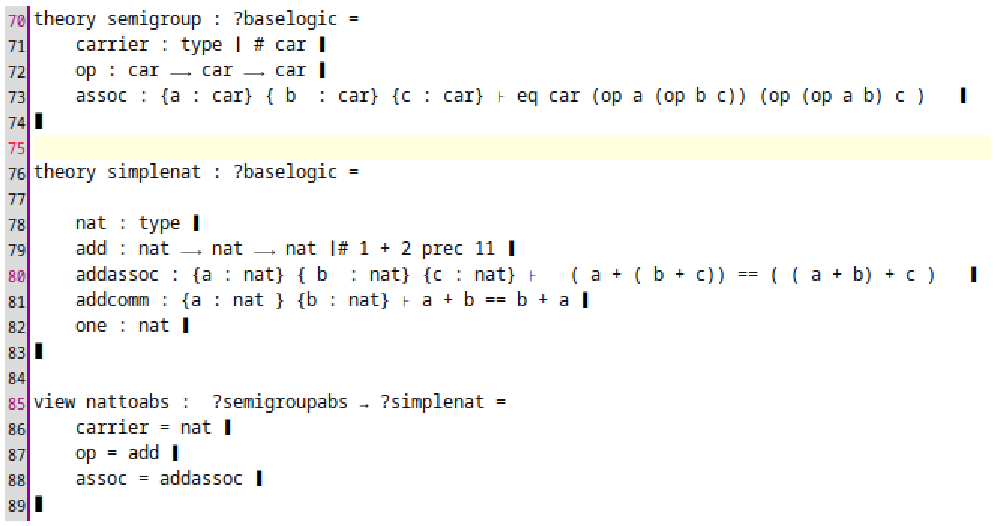
Figure 4: constant with definition



```
{a : type} a → /u a
```

Figure 5: partially inferred type of "example"

Unknowns An unknown in MMT is a named placeholder for a yet to be constructed/found term. Unknowns are never explicitly written in MMT source but rather introduced dynamically when needed (for example during type inference when only a part of the final type is known). For example in figure 4, when MMT infers the type of "example", an intermediate state could look like figure 5 where "/u" is the unknown.
An unknown can depend on a locally introduced variables (variables bound in a Pi or lambda-binder). MMT declares such local dependencies as an application of those variables to the unknown (OMA(nameOfUnknown , OMV(nameOfVariable)*)). The solution for the unknown then declares these local variables as free variables (wrapping the solution in the *free*-binder declares all variables it binds as potentially freely occurring in its body). For example, let */u* be the unknown in the term *[a: boolean ,b : boolean] /u* (where a and b are bound variables). Then the fact that the solution for */u* can depend on a and b is signaled by writing the term as *[a,b] (/u a b)*. Let now the solution for */u* be *f a* where f is a function taking a boolean and returning an element of an arbitrary type. Then the actual solution for */u* would be *free(a,b,f a)*. When this solution now gets inserted into the original term *[a,b] free(a,b,f a) a b* one can now see how the application of the bound variables and the freely "bound" variables match. This principle is



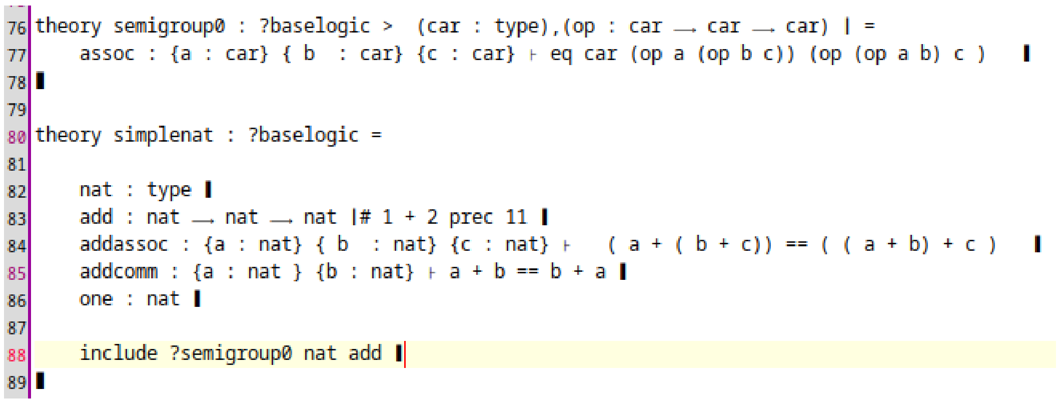
```
theory semigroup : ?baselogic =
    carrier : type ❘ # car ❙
    op : car ⟶ car ⟶ car ❙
    assoc : {a : car} { b  : car} {c : car} ⊦ eq car (op a (op b c)) (op (op a b) c )   ❙
❚

theory simplenat : ?baselogic =

    nat : type ❙
    add : nat ⟶ nat ⟶ nat ❘# 1 + 2 prec 11 ❙
    addassoc : {a : nat} { b  : nat} {c : nat} ⊦   ( a + ( b + c)) == ( ( a + b) + c )   ❙
    addcomm : {a : nat } {b : nat} ⊦ a + b == b + a ❙
    one : nat ❙
❚

view nattoabs :  ?semigroupabs → ?simplenat =
    carrier = nat ❙
    op = add ❙
    assoc = addassoc ❙
❚
```

Figure 6: A simple abstract theory about semi-groups interpreted in the context of another theory

very important for understanding the concrete implementation of the prover system in later chapters.

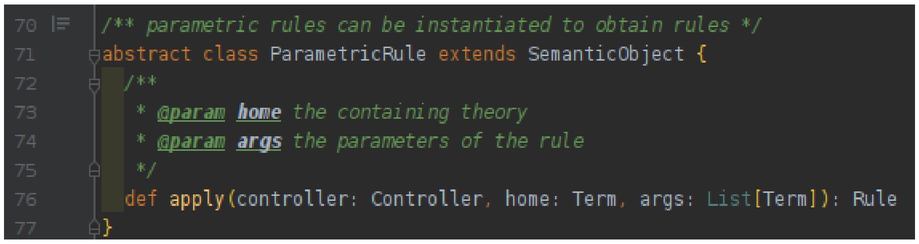
Views One of the more unique features of MMT are the views. A view is theory-mapping that maps elements from the source theory to the target theory, meaning that it shows how a theory (the source) is interpreted in a target theory. This can also be used to define an abstract theory and then interpreting it in a concrete context. Figure 6 shows an example of how to interpret an abstract semi-group theory in a theory that (pretends to) implement the natural numbers. To create a view one simple starts by using the *view*-keyword, followed by a name and then stating the source theory (lhs of the arrow) followed by the target theory. Then one lists how an element of the source theory is interpreted in the target theory. It is also possible to have partial view, i.e. one can leave out how something from the source is interpreted in the target theory. MMT will then generate a warning that some of the "translation" is missing.

Parametreized theories Another feature of MMT, although not unique, is that one can give theories parameters. This lets one define abstract theories which can then be instantiated with concrete parameters. Using the semi-



```
theory semigroup0 : ?baselogic >  (car : type),(op : car ⟶ car ⟶ car) ❘ =
    assoc : {a : car} { b  : car} {c : car} ⊦ eq car (op a (op b c)) (op (op a b) c )   ❙
❚

theory simplenat : ?baselogic =

    nat : type ❙
    add : nat ⟶ nat ⟶ nat ❘# 1 + 2 prec 11 ❙
    addassoc : {a : nat} { b  : nat} {c : nat} ⊦   ( a + ( b + c)) == ( ( a + b) + c )   ❙
    addcomm : {a : nat } {b : nat} ⊦ a + b == b + a ❙
    one : nat ❙

    include ?semigroup0 nat add ❙
❚
```

Figure 7: The example from figure 6 rewritten using a parameterized theory

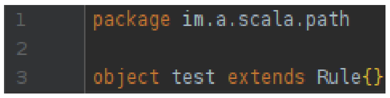


```
/** parametric rules can be instantiated to obtain rules */
abstract class ParametricRule extends SemanticObject {
  /**
   * @param home the containing theory
   * @param args the parameters of the rule
   */
  def apply(controller: Controller, home: Term, args: List[Term]): Rule
}
```

Figure 8: The abstract class defining the parametric rule

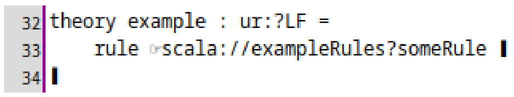
group example once again and rewriting it to use a parameterized theory instead of a view the example in figure 6 can be rewritten as depicted in figure 7.

Rules Rules in MMT are a mechanism that allows a user to extend MMT in several different ways. Among those are: How a term gets handled during type checking/type inference, parser modifications, tactic behavior etc. Rules come in two flavors: Basic rules and parameterized rules.
A rule is implemented as a piece of Scala code. A rule has to implement the rule-trait.



```
package im.a.scala.path

object test extends Rule{}
```

Figure 9: Trivial example Rule



```
theory example : ur:?LF =
    rule ☞scala://exampleRules?someRule ❙
❚
```

Figure 10: How a rule is included inside an MMT-file (concrete example)

rule
jURI
URI(path to rule)

Figure 11: Abstract description how a rule is imported

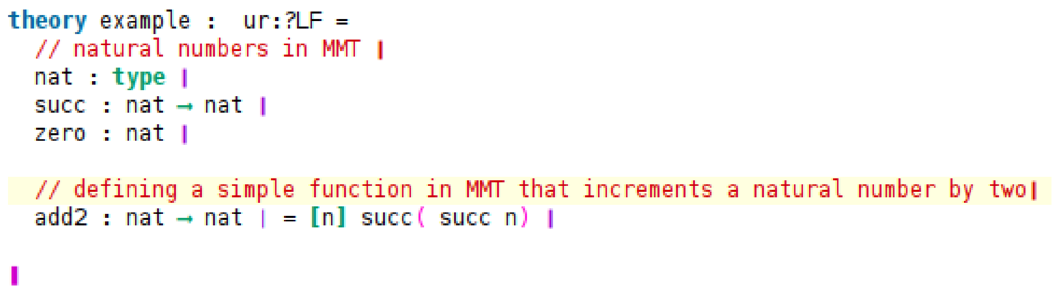
Rules are included as a declaration inside an MMT-file by starting with the *rule* keyword, followed by the *jURI*-unicode and then the actual URI. The URI itself is build by translating the Scala-path into an URI, starting with "scala://" followed by the translated path (replacing the dots in the Scala path with back slashes, followed by a "?" and the name of the rule-object). For example when a rule-object named "test" resides in the package "im.a.scala.path" (figure 9). Then the URI which references this rule translates to "scala://im/a/scala/path?test".
For MMT to be able to find a rule in that way it is necessary to implement the rule as an object. Once included, a rule is added to the global rule set (literally called *RuleSet*) in which MMT stores all the rules it has loaded.
MMT also offers parameterized rules. These rules aren't really rules on their own since they generate rules based on the parameters passed (but they get called through MMT's rule mechanism but with parameters; they also don't get added to the *RuleSet*). To implement a parametric rule one has to create an object which inherits from the abstract class *Parametricrule* (figure 8). This will then force the user to implement a method which returns a rule. This generated rule will then be included into the global *RuleSet*. There is no need to reference these generated rules inside of an MMT-file. Unlike the parametric rule, or simple rules, the generated rules don't need to be a Scala object.

2.2 LF

The "Logical Framework" [7] (short LF) is a simplistic dependently typed lambda calculus. It is simplistic in the sense that it comes with a minimal set of features and syntax. This scarcity makes it suitable as a meta-logic (a meta-logic should be as unspecific as possible to not force a certain direction).

LF-OBJ	::=	LF-PI | LF-ARROW | LF-CONSTANT | LF-APPLY
LF-PI	::=	OMBINDC(OMID("Pi") , LF-VARDECL* , LF-OBJ)
LF-ARROW	::=	OMA(OMID("arrow"), LF-OBJ LF-OBJ)
LF-LAMBDA	::=	OMBINDC(OMID("lambda") , LF-VARDECL* , LF-OBJ)
LF-CONSTANT	::=	OMV(LOCALNAME) | OMID(uri-string)
LF-APPLY	::=	OMA(OMID("apply") , LF-OBJ LF-OBJ)
LF-VARDECL	::=	VARDECL(string, [LF-OBJ], [LF-OBJ])

Figure 12: LF grammar in MMT (for brevity names like "arrow" are not fully qualified)



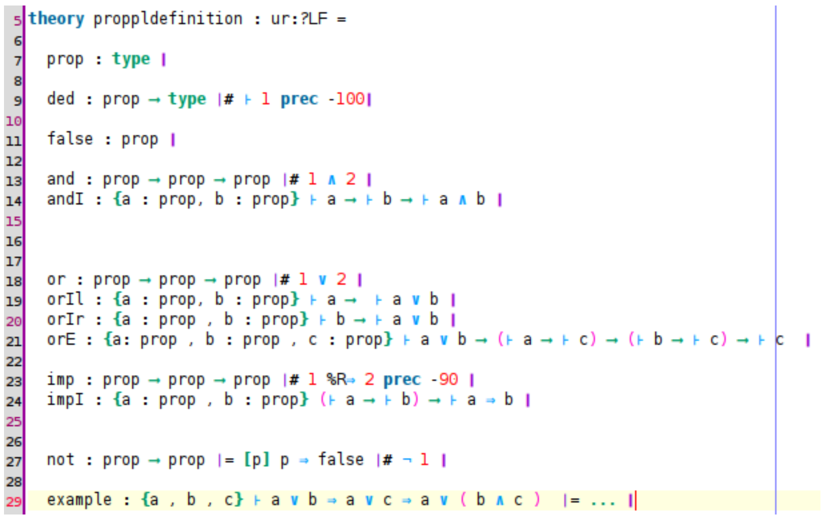
```
theory example :  ur:?LF =
  // natural numbers in MMT ❙
  nat : type ❙
  succ : nat → nat ❙
  zero : nat ❙

  // defining a simple function in MMT that increments a natural number by two❙
  add2 : nat → nat ❘ = [n] succ( succ n) ❙

❚
```

Figure 13: defining a type, it's constructors and a simple function working on that type

The most notable missing features are pattern matching and inductive data types which are otherwise ubiquitous in functional languages. Instead, data types and their constructors are given axiomatically. Function definitions, as long as they don't need pattern matching can be given as plain lambda terms.

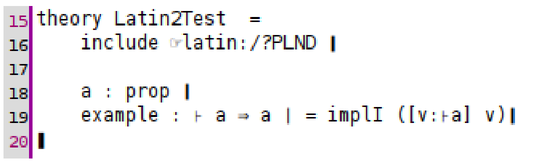
Representation of axioms and theorems in LF A constant in MMT/LF is an axiom by giving it a type but not an accompanying definition. Theorems are merely explicitly typed definitions (by Curry-Howard a theorem is true if its type is inhabited).

Examples Figure 13 shows how a new type and a simple function would be defined in MMT using LF as meta theory. Nat has two constructors, *succ* for successor which essentially increments the value of a natural number by one and *zero*. The function *add2* increments the value of a natural number



```
theory proppldefinition : ur:?LF =

  prop : type ❙

  ded : prop → type ❘# ⊦ 1 prec -100❙

  false : prop ❙

  and : prop → prop → prop ❘# 1 ∧ 2 ❙
  andI : {a : prop, b : prop} ⊦ a → ⊦ b → ⊦ a ∧ b ❙



  or : prop → prop → prop ❘# 1 ∨ 2 ❙
  orIl : {a : prop, b : prop} ⊦ a →  ⊦ a ∨ b ❙
  orIr : {a : prop , b : prop} ⊦ b → ⊦ a ∨ b ❙
  orE : {a: prop , b : prop , c : prop} ⊦ a ∨ b → (⊦ a → ⊦ c) → (⊦ b → ⊦ c) → ⊦ c  ❙

  imp : prop → prop → prop ❘# 1 %R⇒ 2 prec -90 ❙
  impI : {a : prop , b : prop} (⊦ a → ⊦ b) → ⊦ a ⇒ b ❙


  not : prop → prop ❘= [p] p ⇒ false ❘# ¬ 1 ❙

  example : {a , b , c} ⊦ a ∨ b ⇒ a ∨ c ⇒ a ∨ ( b ∧ c )  ❘= ... ❙
```

Figure 14: embedding of propositional logic in MMT



```
theory Latin2Test  =
    include ☞latin:/?PLND ❙

    a : prop ❙
    example : ⊦ a ⇒ a ❘ = implI ([v:⊦a] v)❙
```

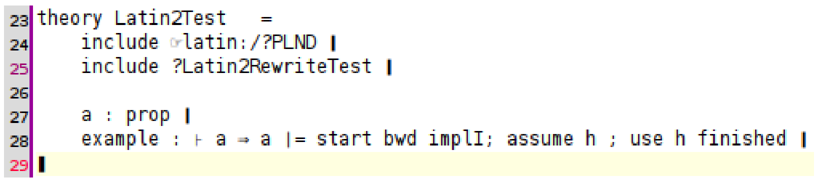
Figure 15: Simple theorem in MMT

by two.
Figure 14 displays a way one could implement propositional logic in MMT. The theory contains the theorem *example* which will be used as an example throughout this work (and will eventually be proven in the tutorial in appendix A).

In case one wishes to parse something the deviates from this scheme MMT offers custom parsing extensions. Such extensions can be included via the rule mechanism MMT offers.

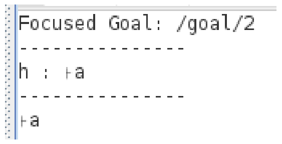
2.3 Proofs in MMT [31]

A theorem in MMT/LF is done by expressing a statement as a type to a constant. Classically the proof then is done by giving a definition that matches the given type.
In fig. 15 the definition *example* represents the trivial theorem that a proof



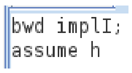
```
theory Latin2Test   =
    include ☞latin:/?PLND ❙
    include ?Latin2RewriteTest ❙

    a : prop ❙
    example : ⊦ a ⇒ a |= start bwd implI; assume h ; use h finished ❙
❚
```

Figure 16: Simple tactics proof in MMT



```
Focused Goal: /goal/2
---------------
h : ⊦a
---------------
⊦a
```

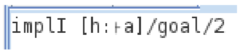
Figure 17: Intermediate proof state of example



```
bwd implI;
assume h
```

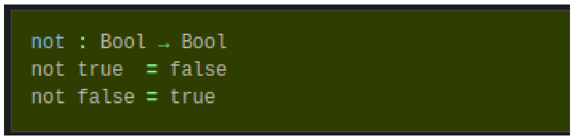
Figure 18: Tactics entered so far for the proof of *example*

of a implies a proof of a.
An alternative way to specify a definition/proof is to use tactics. An example of a proof done using tactics is given in fig. 16. Instead of specifying the lambda term directly, the individual proof steps are listed. The proof then gets turned into a lambda term which gets checked against the type of *example*. Writing a proof using tactics requires the user to specify the whole proof, otherwise the type checking will fail.
Since writing lambda terms and tactic proofs in the above stated fashion for non trivial proofs is fairly hard to achieve MMT offers an interactive mode for writing tactic proofs in a step by step fashion (currently only available for the JEdit plugin for MMT). The advantage of using the interactive mode (for writing tactic proofs) is that the user can inspect the proof state (fig. 17), the build lambda term including it's holes (fig. 19), the entered tactics so far (fig. 18) and error messages.
A simple proof and an overview of available tactics is given in appendix A.



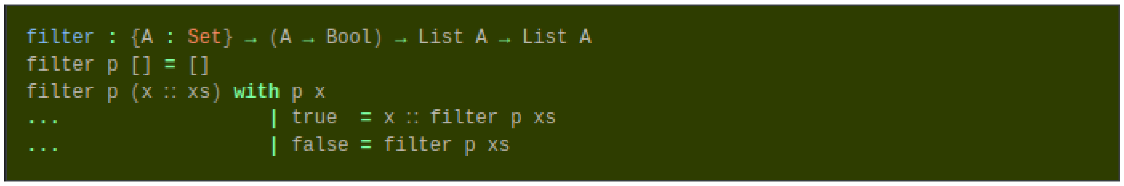
```
implI [h:⊦a]/goal/2
```

Figure 19: The generated lambda term that represents the tactics proof so far *example*



```
not : Bool → Bool
not true  = false
not false = true
```

Figure 20: Simple function in Agda, it implements the unary boolean *not*-operator



```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
...                  | true  = x :: filter p xs
...                  | false = filter p xs
```

Figure 21: with abstraction in Agda

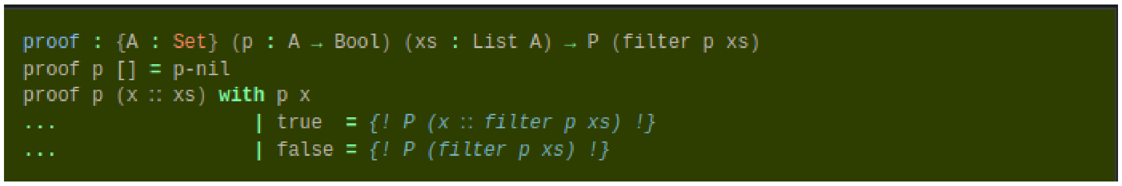
3 Related work

3.1 Agda

Agda is a dependently typed functional programming language implemented in Haskell [10]. It's syntax is designed in a way that is very close to Haskell's syntax. Agda was designed with a strong focus on programming.
Proofs in Agda are given by constructing a lambda term. Agda offers some interactive support to ease the burden of writing proofs in this style. Proofs can be constructed in several steps. This is done by leaving so called holes (unknowns in MMT) for which Agda will infer the type and context.

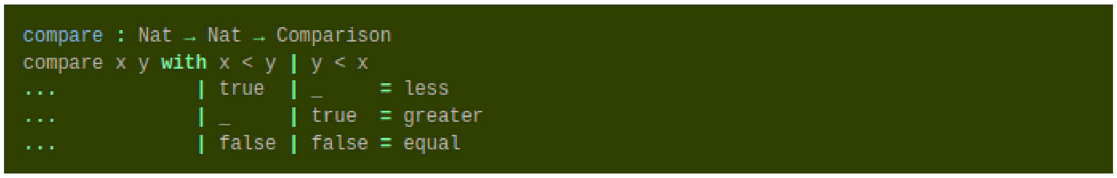
Function definition and pattern matching in Agda A function in Agda is defined by giving a type which is then followed by so called clauses. A clause consists of the function being applied to its arguments, which can be pattern matched, followed by the equals sign and the function body [11]. A simple example function that implements the boolean *not*-function is given in 20.

With-Abstraction With abstraction is a feature of Agda that lets one match rhs of the equal sign in a function clause [15].
The power of the with abstraction comes from the fact that its matching results generalize over types of other terms (potentially including the



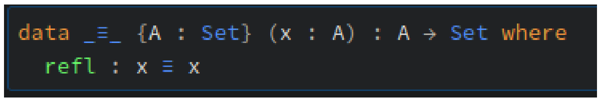
```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p [] = p-nil
proof p (x :: xs) with p x
...                 | true  = {! P (x :: filter p xs) !}
...                 | false = {! P (filter p xs) !}
```

Figure 22: with abstraction in Agda



```
compare : Nat → Nat → Comparison
compare x y with x < y | y < x
...            | true  | _     = less
...            | _     | true  = greater
...            | false | false = equal
```

Figure 23: simultaneous with-abstraction



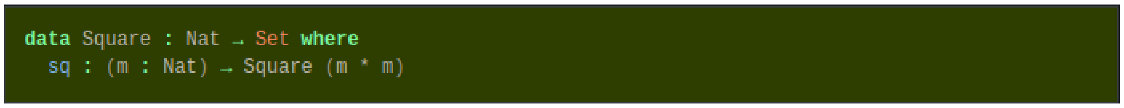
```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Figure 24: Definition of equality in Agda

goal/return type) in the same local context. In the example in figure 22 the match of *p x* generalizes over the type of the goal, *P (filter p (x::xs))*, resulting in *x :: filter p xs* and *filter p xs* respectively .
It is also possible to have multiple *with*-abstraction matches at once as shown in 23.

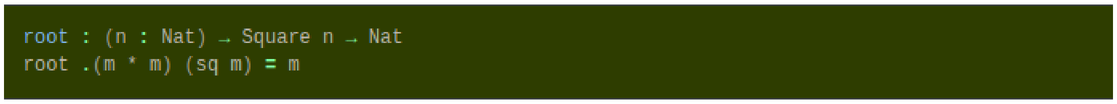
Equality Equality in Agda is defined as a simple inductive type (fig. 24) [26]. The only constructor this type has is a proof that shows that the left hand side is equal to the right hand side.

dot pattern Dot patterns are used when the value for a pattern match is defined strictly by other matches [12]. For example when matching a parameter of type *Square* as in figure 26 it will force the first parameter in that example to be *m * m*. This is in so far special because usually one can only match constructor of a type (in case of *nat* it would be *zero* and *successor*) that is matched but in the special case of dot patterns one can match a complex expression. This behavior is important for rewriting.



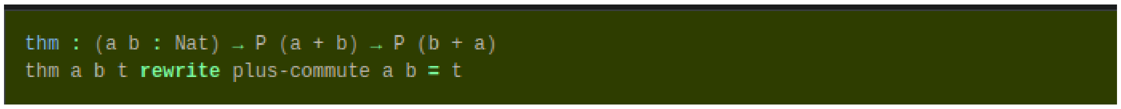
```
data Square : Nat → Set where
  sq : (m : Nat) → Square (m * m)
```

Figure 25: definition of square as data type



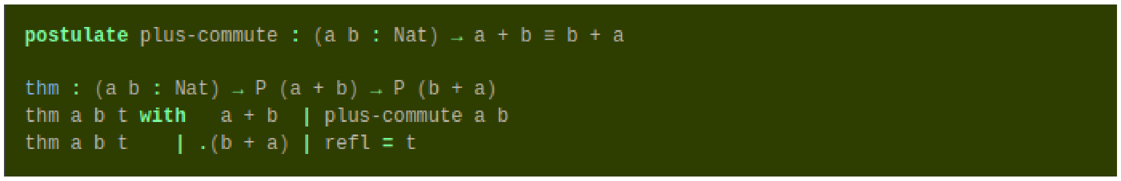
```
root : (n : Nat) → Square n → Nat
root .(m * m) (sq m) = m
```

Figure 26: example of a dot pattern; when the *Square*-parameter is matched it forces the first parameter to be *m * m*



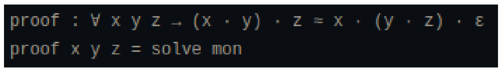
```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t rewrite plus-commute a b = t
```

Figure 27: rewriting syntax in Agda



```
postulate plus-commute : (a b : Nat) → a + b ≡ b + a

thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with   a + b  | plus-commute a b
thm a b t    | .(b + a) | refl = t
```

Figure 28: what a rewrite statement actually translates to in Agda



```
proof : ∀ x y z → (x · y) · z ≡ x · (y · z) · ε
proof x y z = solve mon
```

Figure 29: Example that can be solved with the monoid solver

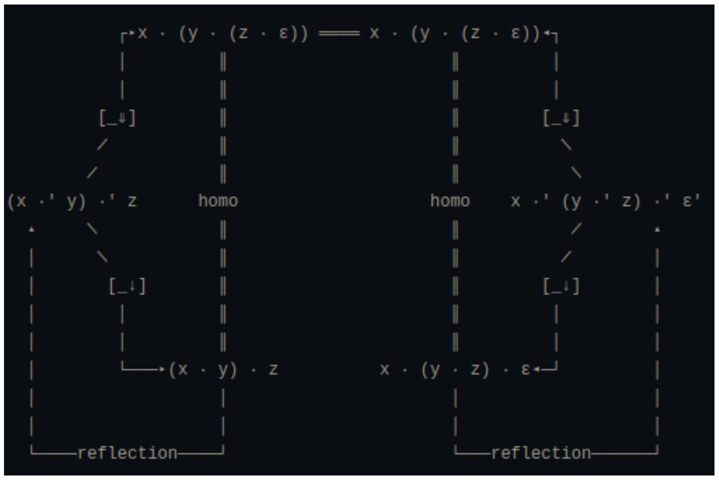


Figure 30: a rough overview over how the monoid solver works on the example in figure 29

rewriting in Agda Rewriting in Agda is implemented by utilizing with-Abstraction [14]. As explained in the previous paragraph one can match complex expressions in case they are forced by another match anyway. This mechanism can be used when matching on something of type *lhs* = *rhs* which will force every occurrence of lhs to be replaced with rhs. Figure 27 shows the rewriting syntax in action and figure 28 shows how this translates into with abstraction. Without the dot-pattern in 28, i.e. matching a constructor or a catch-all variable (for example *ab*) the goal would be either *P zero*, *P (succ n)* (where n is a variable of type nat) or *P ab*. With the dot-pattern match it turns into *P (b + a)*.

monoid and (semi-)ring solver in Agda

monoid solver Agda's solver for monoids is a macro that only works on the conclusion and only on conclusions of shape *lhs* = *rhs* [5]. The approach is, roughly speaking, to translate the goal into its AST-representation using Agda's reflection API (the –reflection– part in the example diagram in 30). Since the solver is written in Agda itself it would not be possible to reason

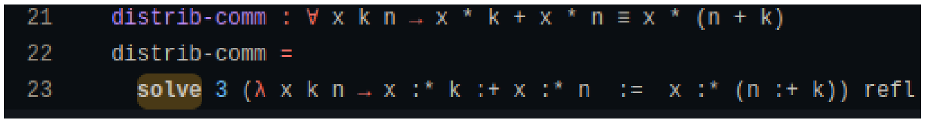

```
distrib-comm : ∀ x k n → x * k + x * n ≡ x * (n + k)
distrib-comm =
  solve 3 (λ x k n → x :* k :+ x :* n  :=  x :* (n :+ k)) refl
```

Figure 31: using the legacy solve tactic to proof a theorem about commutativity and distributivity of multiplication and addition of natural numbers

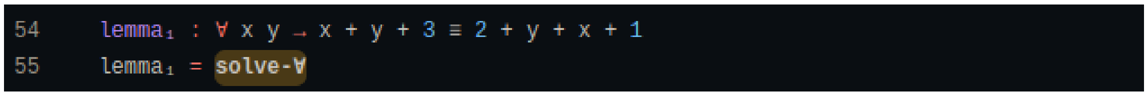

```
lemma₁ : ∀ x y → x + y + 3 ≡ 2 + y + x + 1
lemma₁ = solve-∀
```

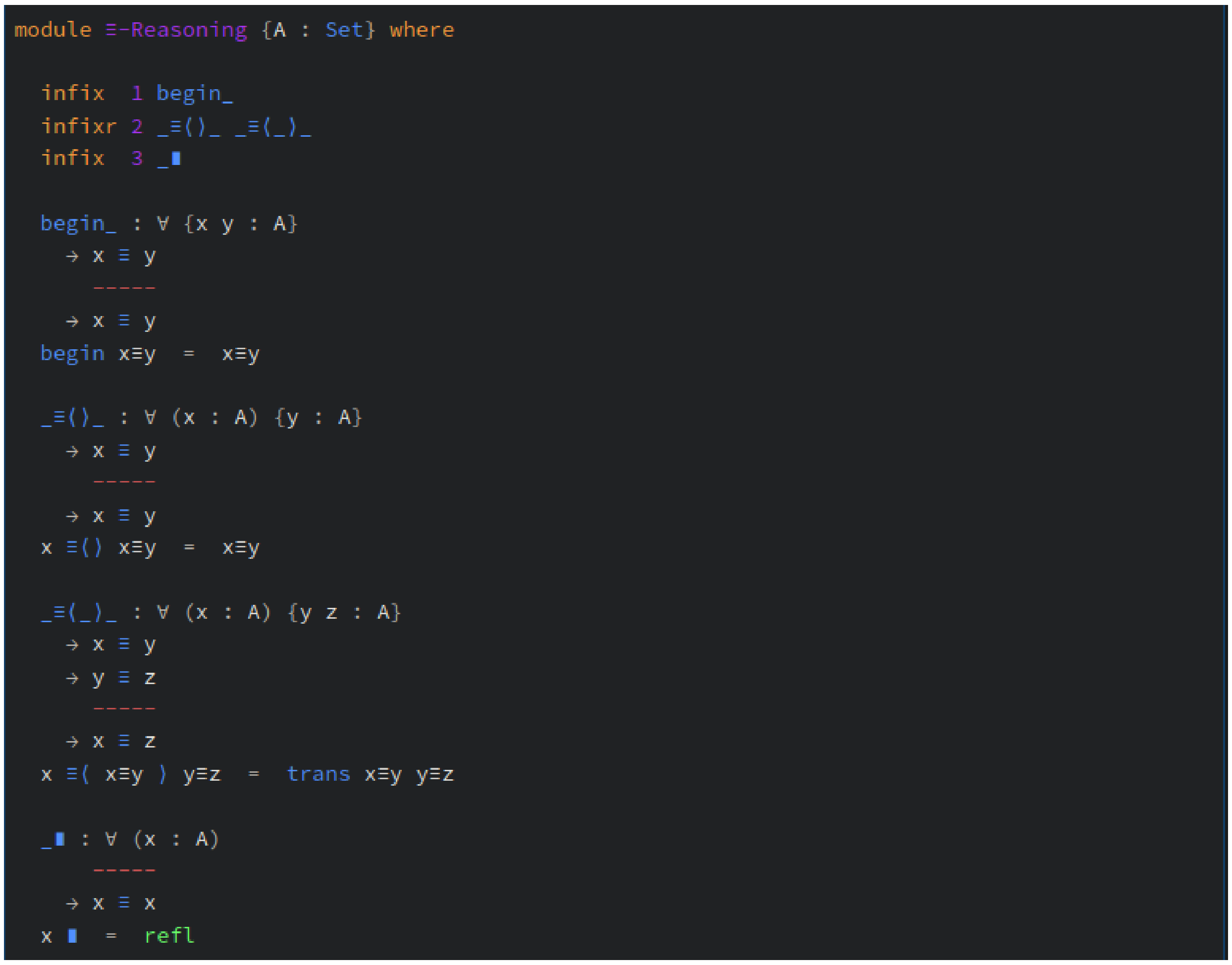
Figure 32: Usage of the fully automatic solve-∀ tactic

about the conclusion without translating the plain term into elements of an inductive data type. After that, the term is normalized and translated back to "normal" Agda. Then the left hand side and right hand side are compared and proven via reflexivity (the === part in the diagram at the top). A proof that normalization is a homomorphism guarantees that the original lhs and rhs are also equal (the ==homo== part in the diagram).
The theorem displayed in figure 30 shall be proven with the monoid solver. To do so one simply includes the appropriate library and calls the solver via *solve* + a parameter which proves that the type the monoid solver tries to solve forms a monoid using the binary operation used in the goal. Under the hood the solver works as depicted in fig. 30. The quoted lhs and rhs of the goal get back translated one time normalized *[_⇓]* and one time without modification *[_↓]*. Then the two normalized terms get compared (in on the very top in figure 30). Through the homomorphism-proof it is guaranteed that the original term and the normalized term are the same and thus that the original lhs and rhs are the same.

(semi-)ring solvers for natural numbers and integer Agda has solvers for (semi-)rings which work on natural numbers and integers. There are the legacy solvers, which require more work from the user, and then there are the "new" solvers which proof a goal fully automatically.
The legacy solvers don't use Agda's reflection library, so the user is required to manually translate the goal into a AST-representation that the legacy tactics can use [1]. The legacy solver tactic takes three arguments: A natural number which indicates how many different variables are in the goal, the translated goal and a proof that solves the remaining sub-goals. The solver

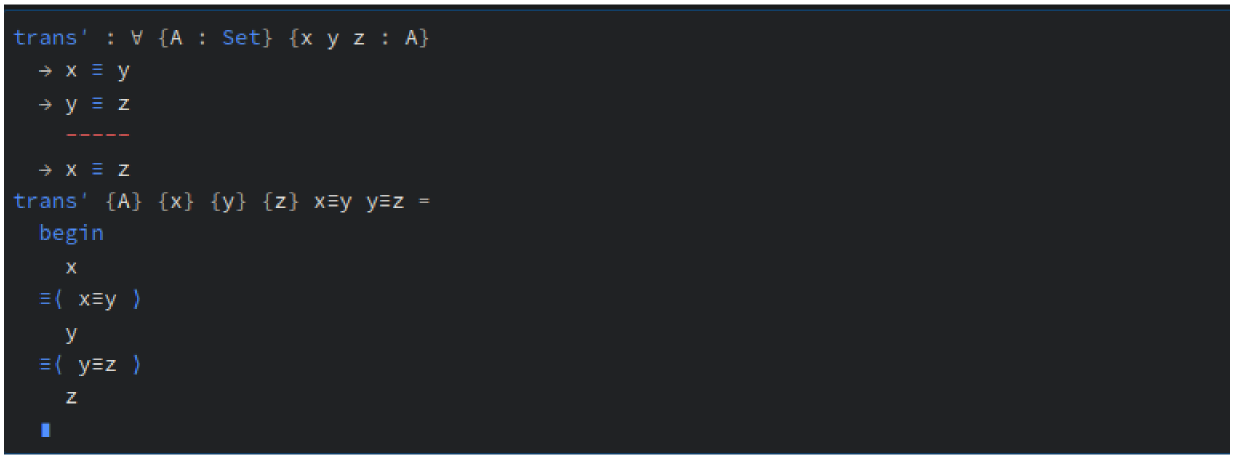
tactic does in fact not solve the goal but tries to transform the *lhs* and *rhs* of the goal so that they are equal. If that fails, the solve-tactic will present the user with new goals. Usually these goals are either to complicated to solve or not solvable at all and the user usually should go back to the original problem and try to modify it (if it is provable). Figure 31 shows a concrete use of the legacy solve tactic. It's first argument indicates that there are three variables (all bound by the forall-quantifier: *x, k, n*). The second argument translates the goal (i.e. the type of *distrib-comm*). A translation is usually straight forward. All variables are bound by a lambda (instead of the forall from the goal) and then the goal is translated almost one to one to the representation syntax. Only the operators are change syntactically by prefixing them with a colon. In this case after the solver has finished its task it presents the user with a new trivial goal that can easily be proven by reflexivity.
Since the usage of the legacy solver tactic is somewhat inconvenient (especially when translating larger terms into its representation syntax), Agda developed a new version of the tactic that is a complete "push button"-solution and does not require any initial setup from the user what so ever [2]. One simply calls the tactics (which is called ∀-solve) and then the tactic either succeeds or fails. Figure 32 shows the usage of the ∀-solve tactic.

chained equality reasoning Agda offers a special syntax for reasoning about equality transitivity chains to represent such chains in a nice readable way [26]. An example of an equality chain would be *(a + b) * c = (b + a) * c = a * c + b * c* . An example of said syntax is given in figure 36. The implementation of it is done in Agda itself. The syntax consists of four elements displayed in figure 33. The **begin**-element is merely for syntax and basically functions as the opening parenthesis. The next element only exists to make something like normalization in a chain more explicit. The next element (_ =< _ > _) is a syntactic representation of the transitivity property of Agda's equality. The last definition ("qed") is a syntactic representation of Agda's equality reflexivity property. This is needed because the transitivity element can't be used directly as first chain element since it needs to take two equality proofs as arguments. One equality proof comes from the argument between the =< and >. The other is supplied the chain part of the chain that comes before the current chain element. But since there is no previous part of the proof chain an "artificial" element is supplied by using reflexivity. The chain in fact is right associative and internally build from right to left



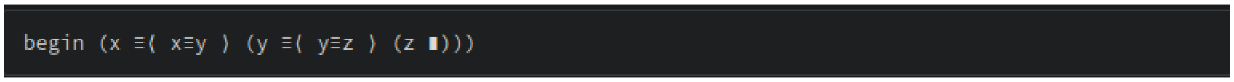
```
module ≡-Reasoning {A : Set} where

  infix  1 begin_
  infixr 2 _≡⟨⟩_ _≡⟨_⟩_
  infix  3 _∎

  begin_ : ∀ {x y : A}
    → x ≡ y
      -----
    → x ≡ y
  begin x≡y  =  x≡y

  _≡⟨⟩_ : ∀ (x : A) {y : A}
    → x ≡ y
      -----
    → x ≡ y
  x ≡⟨⟩ x≡y  =  x≡y

  _≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z  =  trans x≡y y≡z

  _∎ : ∀ (x : A)
      -----
    → x ≡ x
  x ∎  =  refl
```

Figure 33: definition of Agda's chained equality reasoning



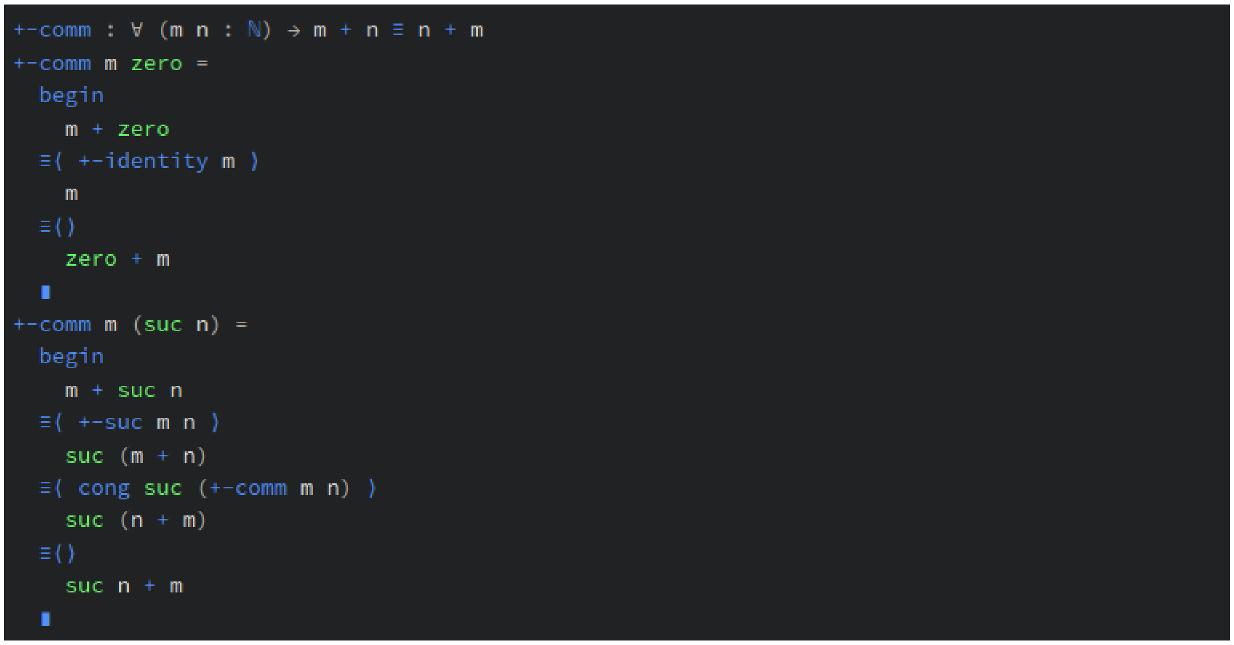
```
trans' : ∀ {A : Set} {x y z : A}
  → x ≡ y
  → y ≡ z
    -----
  → x ≡ z
trans' {A} {x} {y} {z} x≡y y≡z =
  begin
    x
  ≡⟨ x≡y ⟩
    y
  ≡⟨ y≡z ⟩
    z
  ∎
```

Figure 34: the transitivity property expressed a equality chain



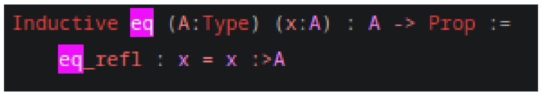
```
begin (x ≡⟨ x≡y ⟩ (y ≡⟨ y≡z ⟩ (z ∎)))
```

Figure 35: how the example in fig. 34 is parsed



```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm m zero =
  begin
    m + zero
  ≡⟨ +-identity m ⟩
    m
  ≡⟨⟩
    zero + m
  ∎
+-comm m (suc n) =
  begin
    m + suc n
  ≡⟨ +-suc m n ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨⟩
    suc n + m
  ∎
```

Figure 36: the transitivity property expressed a equality chain



```
Inductive eq (A:Type) (x:A) : A -> Prop :=
    eq_refl : x = x :>A
```

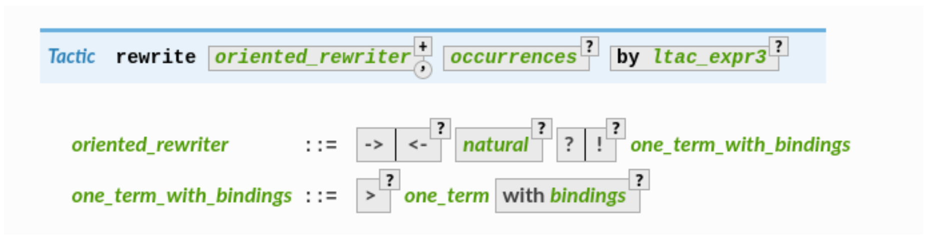
Figure 37: Equality as defined in Coq [6]

even though the natural build progression of the chain would (likely for most users) be from left to right.
A first simple example is displayed in figure 34. As stated above this is implicitly bracketed to the right as displayed in figure 35. First a reflexivity proof is generated for z. This is then passed to the inner chain element as fifth argument (the first being y, the second and third being z, the fourth being the argument named y=z). This part of the chain in turn gets passed to the second chain element and the final chain then gets passed to the **begin** element.
A more complex example is given in fig. 36 proving commutativity of addition defined on natural numbers. This proof works much in the same way as the first example. This time though the purely cosmetic chain element that does not take a proof is used to make the last (when read from left to right) step of the base case and induction step more clear. Internally these steps are not required since Agda only compares normalized terms anyway.

3.2 Coq

Coq is a programming language and proof assistant. In fact Coq itself is made up of two distinct languages: Ltac and Gallina. Gallina is the strict dependently typed programming language that can be used to reason about statements made in it. And then there is Ltac, a impure, script like language, mainly used for creating tactics.
In Coq, proofs can be done in two ways. The more common way to do proofs is via tactics scripts [18] [3] which will indirectly construct a representing lambda term. The less common (but sometimes necessary) way to construct proofs in Coq is by constructing the lambda term directly (more or less similar to Agda).

3.3 Equality reasoning in Coq



```
Tactic rewrite oriented_rewriter+, occurrences? by ltac_expr3?

oriented_rewriter      ::= -> | <-? natural? ? | !? one_term_with_bindings
one_term_with_bindings ::= >? one_term with bindings?
```

Figure 38: rewrite grammar [24]

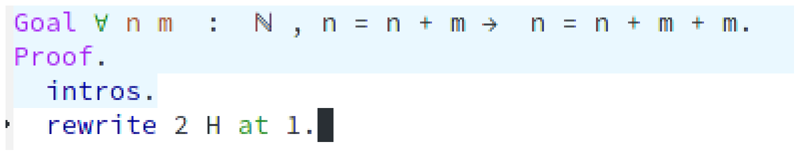
Equality The "build in" equality is defined similar to Agda as an inductive type (see figure 37) where its only constructor is a proof of reflexivity [6].

Rewriting Coq distinguishes between three kinds of equality [23]:

- Leibniz equality
- Setoid equality
- Definitional equality (not important for this work)

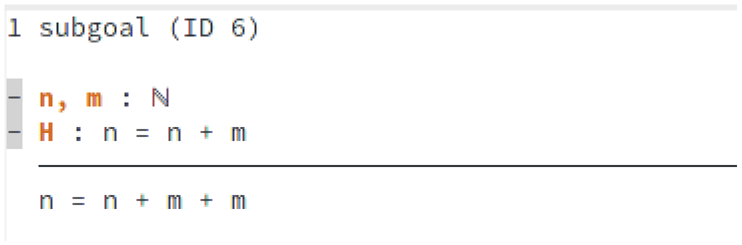
For Leibniz-equality Coq offers the **rewrite**-tactic which is build upon the more general **setoid-reweriting**(see Setoid-Rewriting). A binary relation is considered a Leibniz-equality when it adheres to the property $\{\ x\ y : A\ \}\{P : A \rightarrow Type\}\ x = y \rightarrow P\ x \rightarrow P\ y$ (where A is just some type). Usually for a target relation that property is proven or axiomatically stated somewhere in the context and Coq will find it. In case it is not trivially stated in the context Coq tries to prove it automatically on its own when using the rewrite tactic (usually by "chaining together" other congruence properties for morphisms used in the target term in which the rewriting is taking place).
The rewrite-tactic takes an rewrite relation (f.ex. x = 3) and rewrites a goal accordingly. The **rewrite**-tactic takes multiple optional parameters. An incomplete grammar is given in 38.

The arrows (in oriented_rewriter) indicate the direction in which the equality is to be used (i.e. if $\leftarrow$ is used, symmetry gets applied to the rewrite relation). The **natural**-argument together with the question/exclamation mark modify how often the rewrite is performed. If the question mark together with **natural** is given then the rewrite is performed at most **natural** number of times.*one_term_with_bindings* is the rewrite relation.
occurences specifies the location where the rewriting should take place. All



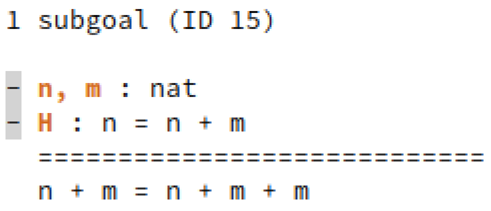
```
Goal ∀ n m  :  ℕ , n = n + m →  n = n + m + m.
Proof.
  intros.
  rewrite 2 H at 1.
```

Figure 39: Example using the rewrite-tactic



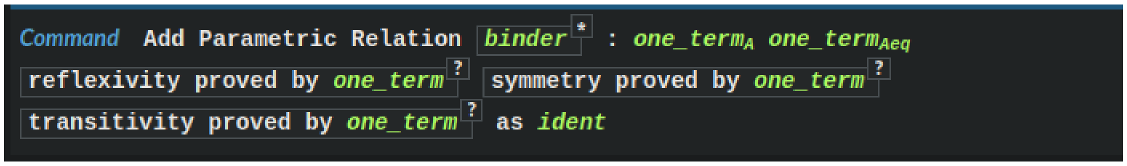
```
1 subgoal (ID 6)

n, m : ℕ
H : n = n + m
______________________________________________
n = n + m + m
```

Figure 40: Proof state of the example in 39 before the rewrite



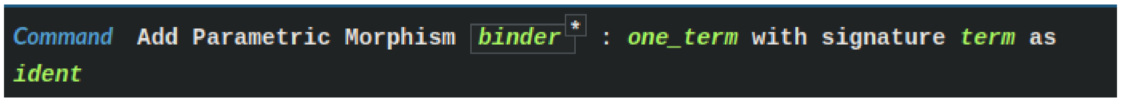
```
1 subgoal (ID 15)

n, m : nat
H : n = n + m
============================
n + m = n + m + m
```

Figure 41: Proof state of the example in 39 after the first rewrite



```
Command  Add Parametric Relation binder* : one_termA one_termAeq
reflexivity proved by one_term? symmetry proved by one_term?
transitivity proved by one_term? as ident
```

Figure 42: Syntax to register a user provided relation for setoid rewriting [21]

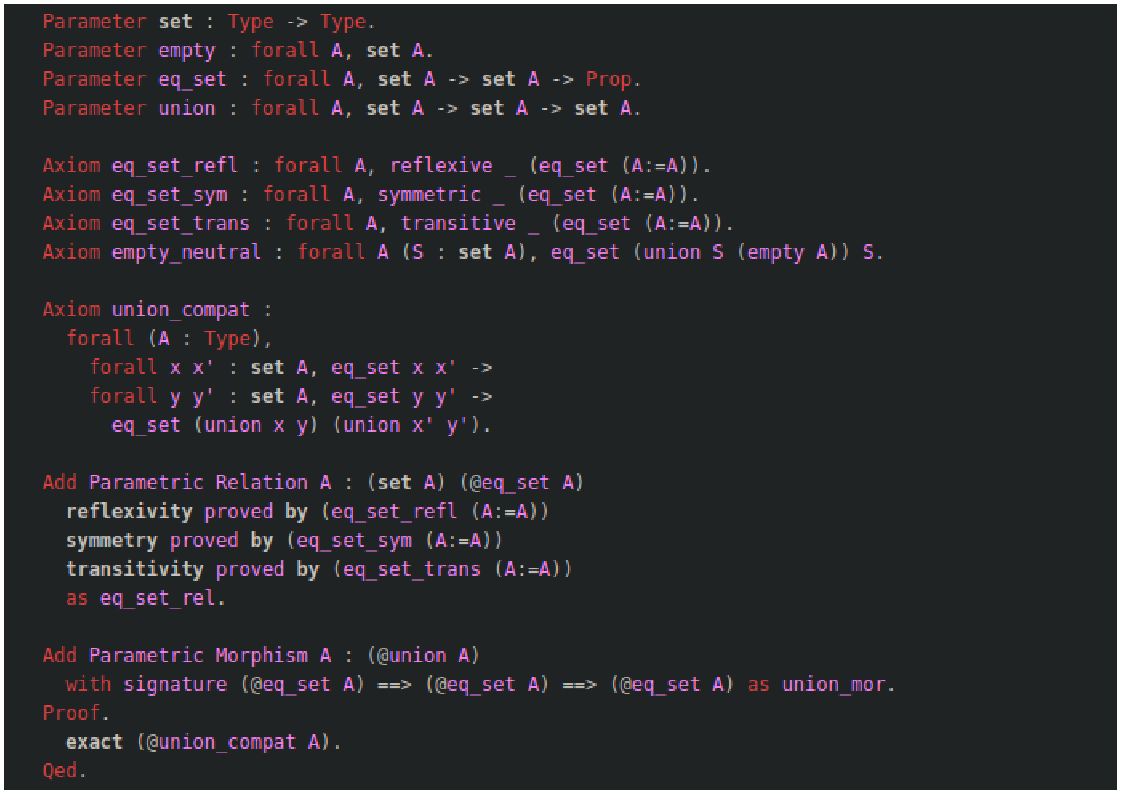


```
Command  Add Parametric Morphism binder* : one_term with signature term as
ident
```

Figure 43: Syntax to register a morphism [20]

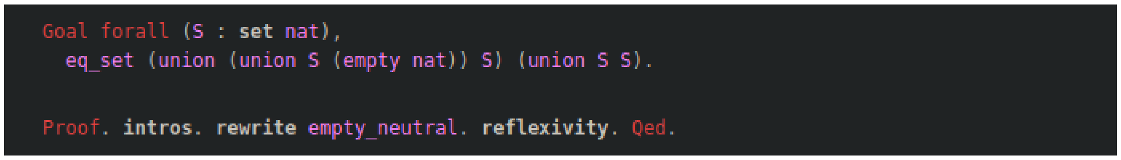
possible sub-expressions which can be rewritten are those who match the left hand side of the rewriting relation. Coq searches for possible rewrites in a depth first search manner, meaning the occurrence parameter selects possible rewrites top to bottom, left to right.
The **by ltac_expr3** parameter applies a tactics expression to potentially newly generated goals. An example using rewrite is given in 39. The hypothesis H is being used twice (first parameter of the rewrite tactic) at the first possible position that can be rewritten. The figures 40 and 41 show the proof state before and after the rewriting.

Setoid-Rewriting Setoid rewriting is a simpler, more general form of the rewrite in the sense that a binary relation can be more likely used for setoid rewriting than the pure rewrite tactic, since setoid rewriting does not require to be congruent towards all morphisms.
To use setoid rewriting with a specific relation it first has to be registered to the Coq system. To do so one uses the **Add** (**Parametric**) **Relation** command as shown in fig. 42 [19]. **one_term$_\mathbf{A}$** is the carrier and **one_term$_\mathbf{Aeq}$** is the user provided relation. Both have to be type-able under the **binder** context. **one_term$_\mathbf{A}$** has to be of type *forall* a_0 a_1 … a_n , *Type*. **one_term$_\mathbf{Aeq}$** has to be of type *forall* a_0 a_1 … a_n , A_0 a_0 a_1 … a_n $\rightarrow$ A_1 a_0 a_1 … a_n $\rightarrow$ … $\rightarrow$ A_m a_0 a_1 … a_n . **ident** is the name which Coq uses to save the morphism to the internal database and is used by Coq for internally used, automatically generated lemmas.
Depending on what proofs have been provided the tactics *symmetry*, *reflexivity* and *setoid_rewrite* (for transitive rewrites in case transitivity has been



```
Parameter set : Type -> Type.
Parameter empty : forall A, set A.
Parameter eq_set : forall A, set A -> set A -> Prop.
Parameter union : forall A, set A -> set A -> set A.

Axiom eq_set_refl : forall A, reflexive _ (eq_set (A:=A)).
Axiom eq_set_sym : forall A, symmetric _ (eq_set (A:=A)).
Axiom eq_set_trans : forall A, transitive _ (eq_set (A:=A)).
Axiom empty_neutral : forall A (S : set A), eq_set (union S (empty A)) S.

Axiom union_compat :
  forall (A : Type),
    forall x x' : set A, eq_set x x' ->
    forall y y' : set A, eq_set y y' ->
      eq_set (union x y) (union x' y').

Add Parametric Relation A : (set A) (@eq_set A)
  reflexivity proved by (eq_set_refl (A:=A))
  symmetry proved by (eq_set_sym (A:=A))
  transitivity proved by (eq_set_trans (A:=A))
  as eq_set_rel.

Add Parametric Morphism A : (@union A)
  with signature (@eq_set A) ==> (@eq_set A) ==> (@eq_set A) as union_mor.
Proof.
  exact (@union_compat A).
Qed.
```

Figure 44: Complete example on how to register a parametric relation and a morphism [19]

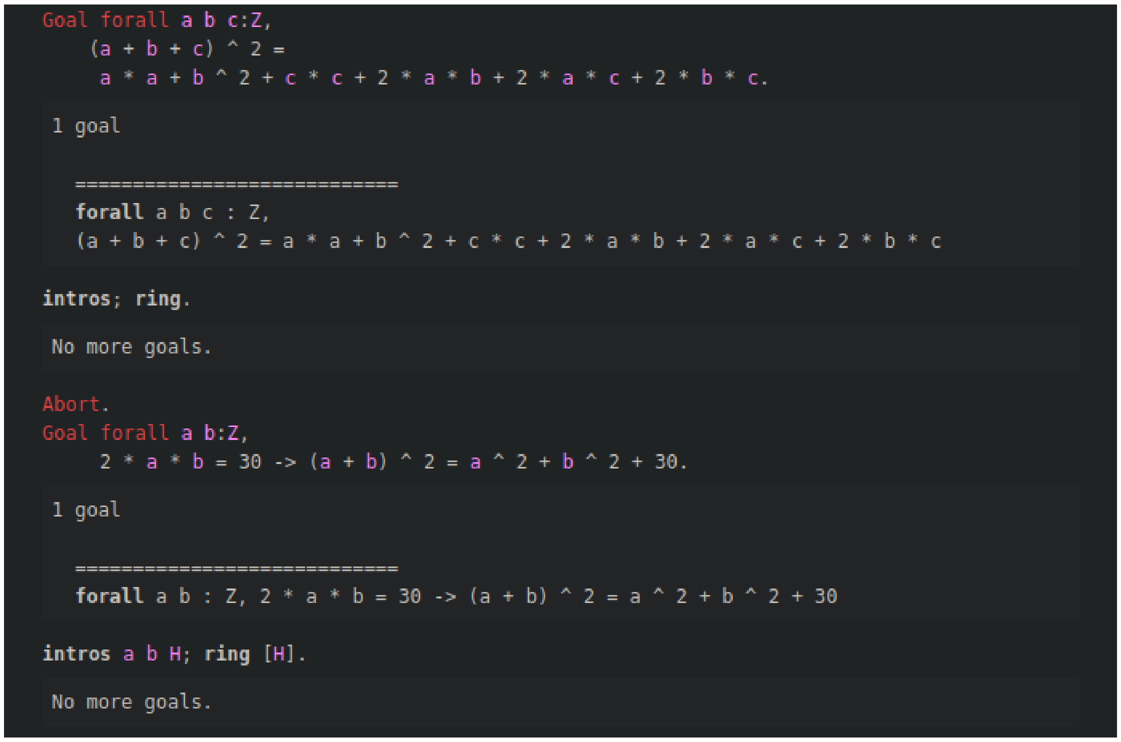


```
Goal forall (S : set nat),
  eq_set (union (union S (empty nat)) S) (union S S).

Proof. intros. rewrite empty_neutral. reflexivity. Qed.
```

Figure 45: Example usage of the registered relation and morphism from fig. 44 [19]

proven) are enabled for the user provided relation. Transitive rewrite means that the setoid_rewrite can only be used for situations where the rewrite results in a transitivity proof. A rewrite step usually results the application of the transitivity property when the goal and the rewrite relation are made up of the same binary relation and the lhs or rhs of the rewrite relation matches the lhs or rhs of the goal. For example, when the rewrite relation is $a = b$ and the goal is $a = c$ then the proof will look something like this $h : a = c \vdash$ *trans a b c h new_goal* where *trans* is the transitivity property of = and *new_goal* is the newly generated goal.
To be able to rewrite in a more complex way one needs to use the **Add** (**Parametric**) **Morphism** (fig. 43). This will be translated by Coq into *term one_term one_term* which the user then has to proof. When Coq's rewrite tactic is called it then considers all morphisms declared that way for rewriting. In general there are two concrete constructs for the signature to use when adding new morphisms for rewriting. The first one being *(rel ==>)^n rel* (see the Coq manual for the definition of ==> also called *respectful*) where rel is a user provided relation and n is the number of arguments **one_term** takes. In short this means that when there is an expression of type *rel a b* (where a and b are arbitrary but fixed terms) in the context and the target expression is of type *rel (one_term args) c* (where *args* are the arguments of *one_term* which includes a) then the target expression can be rewritten to *rel (one_term args0) c* (where *args0* is the same as *args* but with some/all occurrences of a replaced with b). A complete example is given in figure 44 and 45. When *rewrite* is called internally calls setoid_rewrite which then searches for an appropriate theorem that lets the user rewrite under the *union*-morphism.
The second construct for the *signature* in the *Add Morphism*-construct is *(rel ==>)^n (Basics.flip Basics.impl) somenaryproperty* where *somenaryproperty* is a property/function that takes n arguments. This will proof congruence for *rel* with regards to *somenaryproperty.*

ring/field The ring and field tactics can solve equality and inequality statements about structures that form a (semi-)ring/field [22]. The tactics work by simply calling them. The optional arguments that one can pass to these tactics are terms of type $a = b$. (see the example in fig. 46 for two examples of the ring tactic). The passed equalities are used to make the tactic aware of certain selected equalities.


```
Goal forall a b c:Z,
    (a + b + c) ^ 2 =
     a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c.
```

```
1 goal

  ============================
  forall a b c : Z,
  (a + b + c) ^ 2 = a * a + b ^ 2 + c * c + 2 * a * b + 2 * a * c + 2 * b * c
```

```
intros; ring.
```

```
No more goals.
```

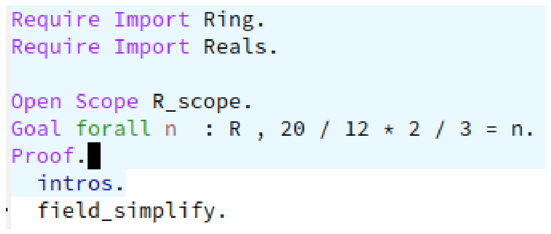
```
Abort.
Goal forall a b:Z,
     2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30.
```

```
1 goal

  ============================
  forall a b : Z, 2 * a * b = 30 -> (a + b) ^ 2 = a ^ 2 + b ^ 2 + 30
```

```
intros a b H; ring [H].
```

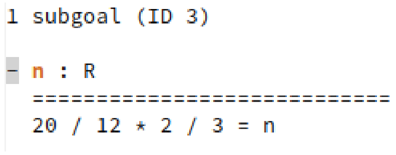
```
No more goals.
```

Figure 46: Example usage of the ring tactic [22]



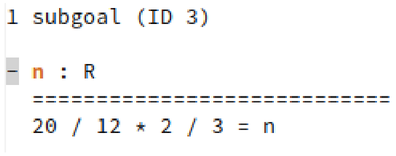
```
Require Import Ring.
Require Import Reals.

Open Scope R_scope.
Goal forall n  : R , 20 / 12 * 2 / 3 = n.
Proof.
  intros.
  field_simplify.
```

Figure 47: Proof using the field_simplify tactic



```
1 subgoal (ID 3)

  n : R
  ============================
  20 / 12 * 2 / 3 = n
```

Figure 48: Proof state before applying the simplification



```
1 subgoal (ID 3)

  n : R
  ============================
  20 / 12 * 2 / 3 = n
```

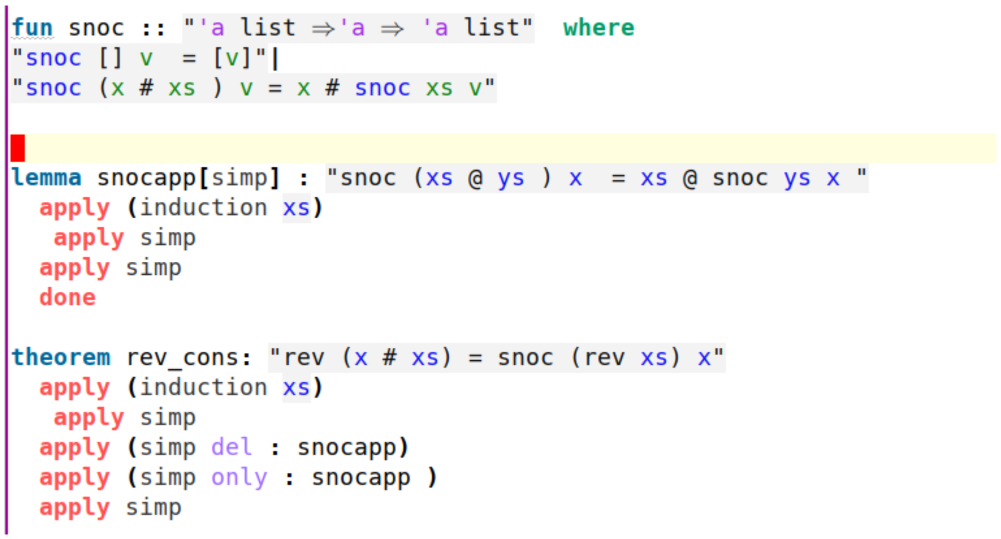
Figure 49: Proof state after applying the simplification

To register a new ring one has to **Add Ring** command which takes a name for the newly registered ring and a proof that shows that the new ring satisfies the ring properties.

ringsimplify/fieldsimplify *ringsimplify* and *fieldsimplify* can simplify expressions of ring/field structures [22]. This tactic can be applied to hypothesis as well. It takes a list of sub-expressions which should be simplified which then get taken by the ring_simplify tactic and replaces the unsimplified version of the sub-expression with the simplified one. If no sub-expressions are specified then the tactic tries to simplified the whole target expression (see example 47 with the proof states 48 and 49).

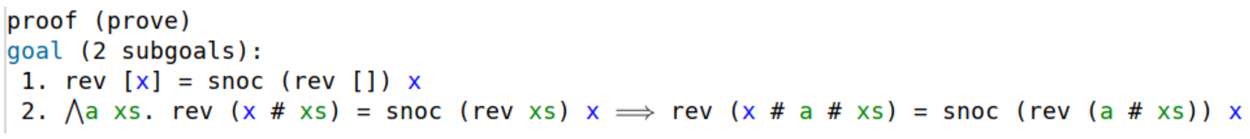
3.4 Isabelle/HOL

Equality and equality reasoning Equality in Isabelle/HOL is defined as boolean equality. Essentially it is a function that takes two terms of the same



```
fun snoc :: "'a list ⇒'a ⇒ 'a list"  where
"snoc [] v  = [v]"|
"snoc (x # xs ) v = x # snoc xs v"


lemma snocapp[simp] : "snoc (xs @ ys ) x  = xs @ snoc ys x "
  apply (induction xs)
   apply simp
  apply simp
  done

theorem rev_cons: "rev (x # xs) = snoc (rev xs) x"
  apply (induction xs)
   apply simp
  apply (simp del : snocapp)
  apply (simp only : snocapp )
  apply simp
```

Figure 50: various uses of *simp*

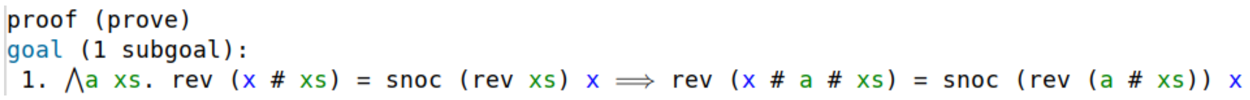


```
proof (prove)
goal (2 subgoals):
 1. rev [x] = snoc (rev []) x
 2. ⋀a xs. rev (x # xs) = snoc (rev xs) x ⟹ rev (x # a # xs) = snoc (rev (a # xs)) x
```

Figure 51: proof state after induction

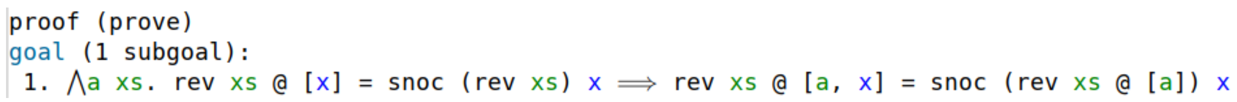
type and returns a boolean [9]. Isabelle offers the common infix notation for equality in form of the equals-sign. When the type of the terms taken by equality is bool then equality is the same as the bi-implication except for the priority (equality has a high priority while iff has a low priority).

simp *simp* is Isabelle's automated method for equality reasoning. This method takes a database of equality rules and applies them left to right to the goal (in some instances simp will automatically apply a rule with the



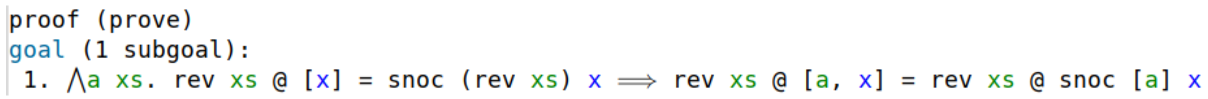
```
proof (prove)
goal (1 subgoal):
 1. ⋀a xs. rev (x # xs) = snoc (rev xs) x ⟹ rev (x # a # xs) = snoc (rev (a # xs)) x
```

Figure 52: before the first application of *simp del : snocapp* in the induction step



```
proof (prove)
goal (1 subgoal):
 1. ⋀a xs. rev xs @ [x] = snoc (rev xs) x ⟹ rev xs @ [a, x] = snoc (rev xs @ [a]) x
```

Figure 53: before the first application of *simp only : snocapp* in the induction step



```
proof (prove)
goal (1 subgoal):
 1. ⋀a xs. rev xs @ [x] = snoc (rev xs) x ⟹ rev xs @ [a, x] = rev xs @ snoc [a] x
```

Figure 54: proof state after *simp only: snocapp*

lhs and rhs flipped) [25]. *simp* offer several modifiers to modify the database (adding deleting etc). *simp* also accepts conditional rewrite rules. In general, a newly proven theorem is added when it is marked with the *[simp]*-flag. Example uses of *simp* are shown in figure 50. The lemma *snocapp* is added to the simp-database indicated by the *[simp]* after the snocapp name. *simp* can not only simplify goals but also solve them. Therefore it was enough in the base case and induction step of the snocapp lemma to use *simp*.
The figures 51 to 54 show the proofs states a various points during the proof of *rev_cons*. Since the induction step does not go through without the *snocapp* lemma the application of *simp del: snocapp* only simplifies the goal, but can not solve it since the modifier *del* removes the said lemma from the simp-database. The next application of *simp* with the *only*-modifier applies only the given set of theorems/lemmas to the goal (in this case only snocapp). The final *simp* then solves the goal.

subst and hypsubst *subst* and *hypsubst* are Isabelle's most basic ways of rewriting. Both can only be used inside an apply-script. *subst* takes a theorem of shape *lhs* = *rhs* and replaces the lhs in the target (which can be the conclusion or a hypothesis) with its rhs [28]. The natural number argument of *subst* tells Isabelle at which position(s) the substitution should take place. *subst* itself can not take a hypothesis as its argument and use it for rewriting/substitution. If one wants to use a hypothesis for rewriting one has to use *hypsubst* which will automatically pick a hypothesis and use it for rewriting/substitution. If *hypsubst* does not pick the desired hypothesis one has to instruct Isabelle with the *back*-command to try another. Overall the controllable of hypsubst is underwhelming.

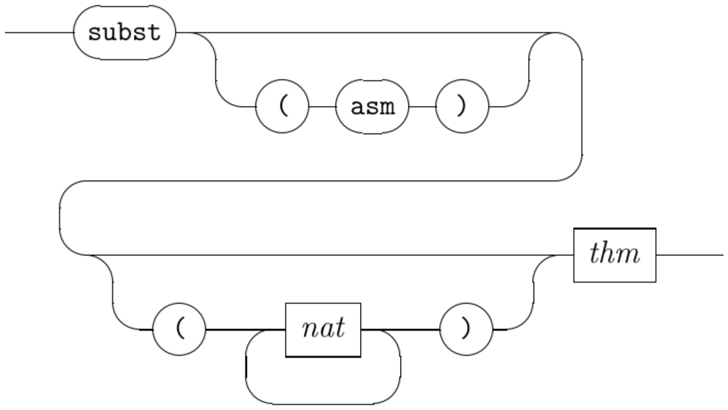
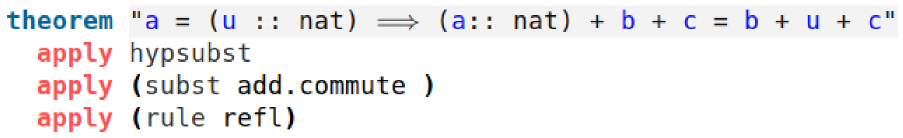
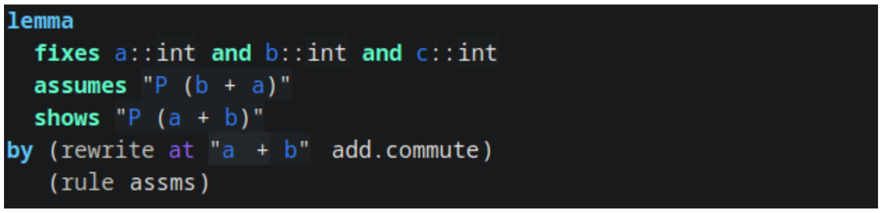


Figure 55: syntax for the subst-tactic [27]



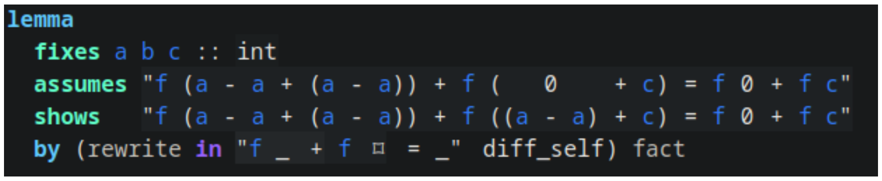
```
theorem "a = (u :: nat) ⟹ (a:: nat) + b + c = b + u + c"
  apply hypsubst
  apply (subst add.commute )
  apply (rule refl)
```

Figure 56: Simple example using *hypsubst* and *subst*



```
lemma
  fixes a::int and b::int and c::int
  assumes "P (b + a)"
  shows "P (a + b)"
by (rewrite at "a + b" add.commute)
   (rule assms)
```

Figure 57: Rewriting at the conclusion



```
lemma
  fixes a b c :: int
  assumes "f (a - a + (a - a)) + f (    0     + c) = f 0 + f c"
  shows   "f (a - a + (a - a)) + f ((a - a) + c) = f 0 + f c"
  by (rewrite in "f _ + f □ = _" diff_self) fact
```

Figure 58: Rewriting at the conclusion; selecting a subterm

The application of *hypsubst* in the example in figure 56 replaces all *a*s with *u*s leading to the new conclusion $u + b + c = b + u + c$. *subst add.commute* then applies commutativity of addition to the goal resulting in the new conclusion $b + u + c = b + u + c$.

Rewriting library The rewriting library for Isabelle is a fairly new development to close the gab between the high level rewriting automation *simp* and the very basic, low level *subst* method [4].
The rewriting library offers ways and means to rewrite (multiple) sub-terms in the goal or in the hypothesis. The rewrite proof method lets the user select a (sub-)term to rewrite using the **at** and **in** keywords then specifying the sub-expression which should be considered for the rewrite. The sub-expression pattern is followed by the actual rewriting relation (see figure 59

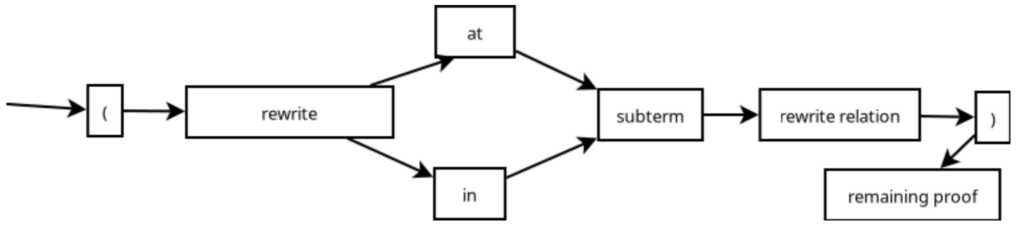


Figure 59: syntax for the rewriting tactic in Isabelle

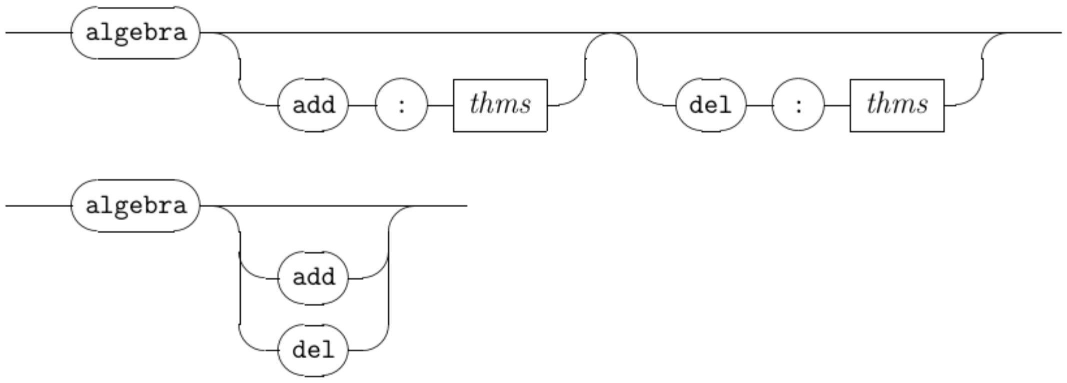
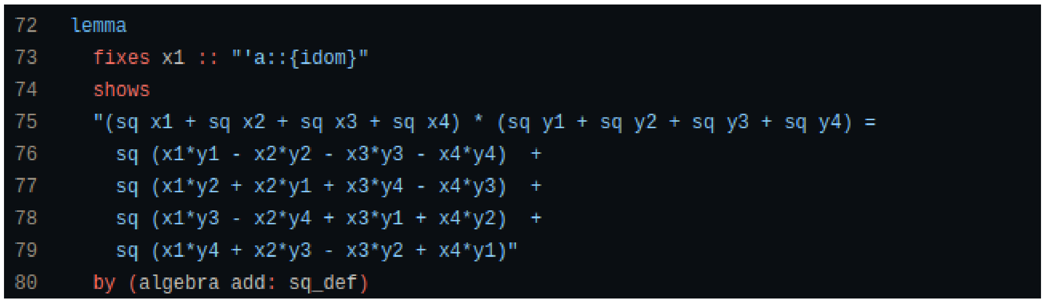


Figure 60: syntax for the algebra proof method [29]

for the syntax). The rewriting relation has to be of the form $\ldots \rightarrow$ *lhs* = *rhs*. In the simple example in figure 57 the "sub"-expression is just the whole goal. The next example in fig. 58 shows that one can leave parts of a sub-expression by using the underscore. Isabelle will then try to match the underscore for anything. Isabelle will search the goal for a (sub-)expression that matches the pattern passed to the rewrite-tactic. Using the little box symbol signals Isabelle where the actual rewrite is supposed to take place in the sub-expression, when a matching one is found. Depending on whether **at** has been used (instead of **in**), the box has match the *lhs* of the rewriting relation. In the example the pattern matches due to its restrictiveness, only the conclusion as a whole. The first underscore matches *a - a + (a - a)*, the second one *f 0 + f c* and the box matches *(a - a) + c*. In case **in** was used the expression that the box matches gets further searched for an expression that matches the *lhs* of the rewriting relation. In the example the rewriting relation (named diff_self) is *v - v = 0* where *v* can be any expression of type *nat*.
This was just a mere peek at what the rewriting library can do, there is in fact more to it like: rewriting in assumption, explicit instantiation of variables in the rewriting relation, rewriting under binders using variable capturing etc. Non the less this should give a rough feeling on what the rewriting library for Isabelle can do.



```
lemma
  fixes x1 :: "'a::{idom}"
  shows
  "(sq x1 + sq x2 + sq x3 + sq x4) * (sq y1 + sq y2 + sq y3 + sq y4) =
    sq (x1*y1 - x2*y2 - x3*y3 - x4*y4)  +
    sq (x1*y2 + x2*y1 + x3*y4 - x4*y3)  +
    sq (x1*y3 - x2*y4 + x3*y1 + x4*y2)  +
    sq (x1*y4 + x2*y3 - x3*y2 + x4*y1)"
  by (algebra add: sq_def)
```

Figure 61: example of using the algebra proof method

Algebraic automation Isabelle offers solver for algebraic automation that involve ring and field structures. The underlying proof method is Hilbert's Nullstellensatz [30]. Its syntax is given in fig. 60. Before the actual algebraic reasoning takes place the algebra proof method performs simplification (using *add* and *del* modify this behavior). The example in fig. 61 shows why a simplification step might be needed, since this proof would not go though without adding the definition of *square* (sq_def) to the algebra tactic (without it the algebra tactic would not be able to unfold the definition of *sq*).

4 Chained transitivity reasoning

4.1 Terminology

- lhs: left hand side of a binary relation; f.ex. in the expression $\mathbf{1} = \mathbf{2}$ is the number one the lhs of the target relation relation
- rhs: right hand side of a binary relation
- Transitivity chain proof: A transitivity proof that is a composition of multiple chained **transitivity chain elements**
- Target binary relation/target relation: the binary relation that is used in a chain step to make a statement of the form *R lhs rhs* (where R is the target relation)
- (Transitivity) chain element/step: A single chain element in a transitivity chain proof consisting of an lhs, rhs and a proof that *R lhs rhs* where R is the target binary relation.

- homogeneous transitivity property is a transitivity property that only relies on one (binary) relation. The classical example of such a property is the transitivity of equality as in if $a = b$ and $b = c$ then $a = c$ (a = b → b = c → a = c)

- heterogeneous transitivity property is a transitivity property that relies on multiple (binary) relations. An example of such a property would be the statement that if $a < b$ and $b = c$ then $a < c$ (a < b → b = c → a < c)

- Multi-step chain proof: A chained transitivity proof that consist of more then one chain element

- inner chain: a chained transitivity proof is a (depending on the implementation) left (or right) recursive construct. This means that a proof is consecutively constructed in the sense that the next step in the proof chain builds upon another inner proof which in turn itself is a chained transitivity proof, the so called inner chain.

- Chaining lemma/scheme: a lemma that tells MMT how to proof that either *R lhs rhs* (where R is the target relation) or in a multi-step chained transitivity proof, how to chain the current chain element to the inner chain

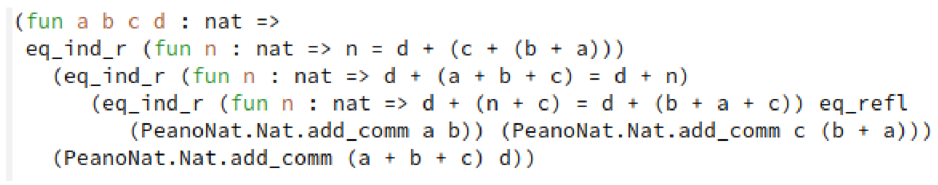
- Local target relation: The binary relation used in a specific chain element. Examples include '=', '<' and '>='.

- Combining proof: a proof that states that *R **lhs rhs*** where R is the **local target relation**

- Linking element: When two chain elements are combined the **lhs** of one chain elements is the **rhs** of the other. Example: lhs << proof >> center << proof0 >> rhs. *center* is the **rhs** of the left chain element and the **lhs** of the right chain element.

4.2 Introduction

Writing long transitivity proofs usually becomes very difficult to get right. As an example consider the term $a+b+c+d = d+(b+(c+a))$ where a, b, c, d are natural numbers and + the left associative, usual operation for addition



```
(fun a b c d : nat =>
 eq_ind_r (fun n : nat => n = d + (c + (b + a)))
   (eq_ind_r (fun n : nat => d + (a + b + c) = d + n)
      (eq_ind_r (fun n : nat => d + (n + c) = d + (b + a + c)) eq_refl
         (PeanoNat.Nat.add_comm a b)) (PeanoNat.Nat.add_comm c (b + a)))
   (PeanoNat.Nat.add_comm (a + b + c) d))
```

Figure 62: pure lambda proof using transitivity statements to proof that $a + b + c + d = d + (b + (c + a))$

$$(((a + b) + c) + d) \overset{\text{commutativity of a and b}}{=} (((b + a) + c) + d) \overset{\text{associativity of b, a and c}}{=} ((b + (a + c)) + d) = ((b + (c + a)) + d) = (d + (b + (c + a)))$$

Figure 63: textbook math proof that chains multiple elements together

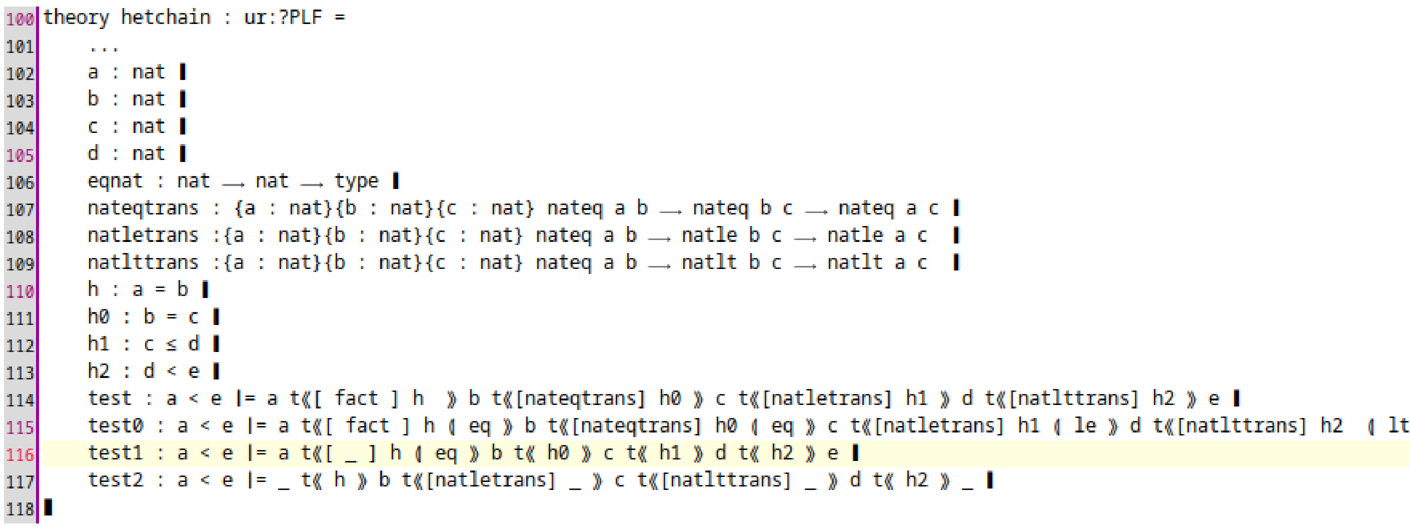
of natural numbers. The main problem with writing pure lambda term proofs that mainly explicitly use transitivity theorems is that their development is syntactically counterintuitive to the way such proof are normally developed (an example of how such a proof looks like when using plain lambda terms is given in 62). In everyday mathematics, when writing proof chains they syntactically resemble fig. 63, where the syntax goes well with the work-flow one would have when proving this statement .

Homogeneous transitivity proofs A proof of transitivity is done by chaining single simpler steps together. One step consists of a statement including the lhs and rhs of an equality statement and the proof that lhs is equal to rhs. A single step itself will then look something like this: *eqchain lhs* $<<$ *proofthatlhsequalsrhs* $>>$ *rhs qed* (there is also another notation/implementation for a transitivity chain element which will be discussed later). Both the "eqchain" and "qed" keywords are required for the proof to be constructed (see the chapter about the implementation why that is). Both are non the less very similar to each other except for two things: which target relations can be used and on the implementation side, how proofs are constructed). Multiple such statements can now be chained together by reusing rhs of one such statement as the lhs of the next statement. Syntactically this will look something like this *lhs* $<<$ *someproof* $>>$ *rhs* $<<$ *anotherproof* $>>$ *nextrhs*. The "anotherproof"-proof will then state that rhs = nextrhs. For a chain greater or equal the size of two the system will then automatically construct a transitivity proof using the proof chain. Returning to the two

step chained proof from before, the system will then construct for a binary transitive relation R with the transitivity property *Rtrans :* $\{a\ b\ c\}\ R\ a\ b \rightarrow R\ b\ c \rightarrow R\ a\ c$, the proof *Rtrans lhs rhs rhs0 someproof anotherproof*. Now when a further chain element is added to the whole chain resulting in something similar to this: *lhs* << *someproof* >> *rhs* << *anotherproof* >> *nextrhs* << *proof3* >> *rhs3*, the system will then first construct a transitivity proof for the two rightmost chain elements (i.e. using "anotherproof" and "proof3") to construct a proof (Rtrans rhs nextrhs rhs3 anotherproof proof3) which in turn then gets used by a second transitivity proof using *lhs* and *someproof* resulting in the final proof *Rtrans lhs rhs rhs3 someproof (Rtrans rhs nextrhs rhs3 anotherproof proof3)*. In general transitivity chain proofs are generated from right to left.
The required proof for each of those statements can be specified with a varying degree of explicitness. Depending on whether the system is able to automatically find a proof, partially infer parts of the proof or is unable to find a proof for the stated equality, the required proof for lhs = rhs can be omitted or at least partially omitted or has to be specified completely in case the system is unable to not even partially infer parts of the proof. Further more, the first lhs and the last rhs in the chain can be omitted in case the type for the constant which holds the proof is provided. Depending on whether a proof is provided or not the lhs or rhs of a chain element can also be omitted (see example 71).
Homogeneous equality chain reasoning can expressed in two ways: The MMT-only way and the version that utilizes the MMT-rule-mechanism. The main difference between both, from a users perspective is how one tells the system which target binary relation (like equality) is to be used for the transitivity chain and which binary relations are even allowed to be used in chained transitivity reasoning. To be able to use MMT-only implementation one has to instantiate the parameterized MMT-theory in which the chained transitivity is defined. Figure 66 show this very theory. To instantiate this theory successfully one has to specify a carrier (i.e. the type on which the target relation is specified on), the target relation, a proof that this relation is transitive and reflexive. After successful inclusion/instantiation one can use the constructs (i.e. the mechanisms and their syntax) for the specified target relation in the current theory. Further more this will also enable the user to import some additional theorems/lemmas about chained transitivity reasoning adapted to the specified target relation.
The other implementation of the homogeneous chained transitivity reason-

ing uses MMT's rule system. Unlike the MMT-only implementation this one is not instantiated by including a parameterized theory. To add a certain binary relation to the set of viable target relations one first has initialize the rewriting mechanism at least partially for the a chosen target binary relation (see the next chapter about rewriting on how one does this). In contrast to the MMT-only implementation, this one requires the user only to specify a carrier, a binary relation and a proof that the specified binary relation is transitive but it is not necessary to provide a proof that the relation is reflexive. Adding a proof that the relation is reflexive and/or the symmetric will allow the system find more theorems for automatic inference of unspecified proofs in the proof chain. When the system tries to find a proof for unspecified proofs in a chain element it goes through a list of available theorem and compares the type of the theorems to the required one. When the symmetry property holds for the target relation, the subroutine that searches for a fitting theorem to fill in as proof will try to apply symmetry to the theorems checked. For example a proof for $R\ a\ b$ is needed in the chain element a << _ >> b but the only available theorem states that $R\ b\ a$. Now with symmetry the system can automatically generate the new corollary that $R\ a\ b$ from the theorem that $R\ b\ a$. Without symmetry the system would not be able to find a proof that $R\ a\ b$. When a proof for $R\ a\ a$ is required and the reflexivity property holds true for R then the system will automatically infer the required proof.
A syntactic difference between the two versions is that the MMT-rule version does not require the beginning "eqchain" and ending "qed" keywords used in the MMT-only version.

Heterogeneous transitional proofs A modification of the above system (especially the implementation of homogeneous chained transitivity proofs that uses the MMT-rule-mechanism) is to generalize the transitivity proof in so far that each chain element can make a transitivity statement using a different transitive relation. An example would be $a = b < c \leq d < e = f$, which would result in the statement that a is less then f (a < f). Other then the homogeneous chained transitivity proofs, there is only one version for the heterogeneous chained transitivity proofs.
The heterogeneous chained transitivity proofs are done similarly by chaining single chain elements together. A single chain element consists of an lhs, rhs and a proof that *R lhs rhs* where R is a binary relation (not necessarily



```
theory hetchain : ur:?PLF =
    ...
    a : nat ❙
    b : nat ❙
    c : nat ❙
    d : nat ❙
    eqnat : nat ⟶ nat ⟶ type ❙
    nateqtrans : {a : nat}{b : nat}{c : nat} nateq a b ⟶ nateq b c ⟶ nateq a c ❙
    natletrans :{a : nat}{b : nat}{c : nat} nateq a b ⟶ natle b c ⟶ natle a c  ❙
    natlttrans :{a : nat}{b : nat}{c : nat} nateq a b ⟶ natlt b c ⟶ natlt a c  ❙
    h : a = b ❙
    h0 : b = c ❙
    h1 : c ≤ d ❙
    h2 : d < e ❙
    test : a < e |= a t⟪[ fact ] h  ⟫ b t⟪[nateqtrans] h0 ⟫ c t⟪[natletrans] h1 ⟫ d t⟪[natlttrans] h2 ⟫ e ❙
    test0 : a < e |= a t⟪[ fact ] h ⦇ eq ⟫ b t⟪[nateqtrans] h0 ⦇ eq ⟫ c t⟪[natletrans] h1 ⦇ le ⟫ d t⟪[natlttrans] h2  ⦇ lt
    test1 : a < e |= a t⟪[ _ ] h ⦇ eq ⟫ b t⟪ h0 ⟫ c t⟪ h1 ⟫ d t⟪ h2 ⟫ e ❙
    test2 : a < e |= _ t⟪ h ⟫ b t⟪[natletrans] _ ⟫ c t⟪[natlttrans] _ ⟫ d t⟪ h2 ⟫ _ ❙
❙
```

Figure 64: transitivity chain reasoning using the heterogeneous implementation

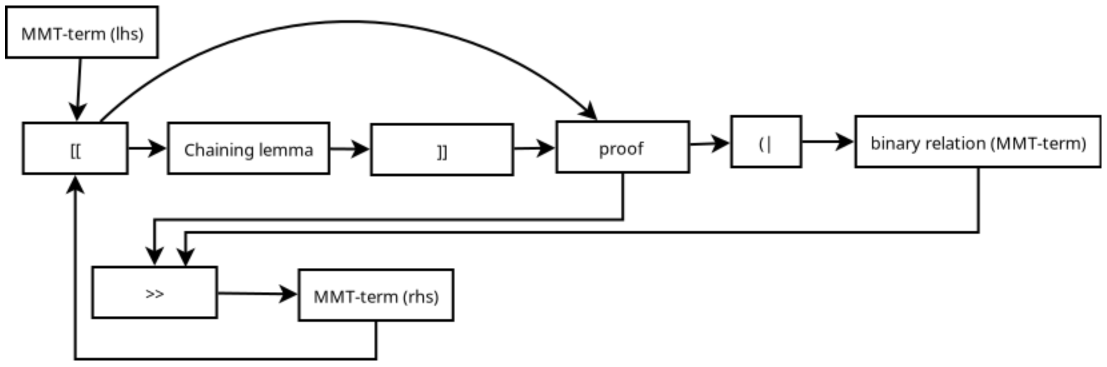


Figure 65: syntax for the the heterogeneous chained transitivity proof

transitive). A single chain element will then look something like this: *lhs [[proof >> rhs* . The elements *lhs*, *rhs* and *proof* can be specified again in varying degrees of explicitness (see example 64) and the system tries to figure them out on its own. When it can't do so the system will throw an error.
When one has two chain elements where the *rhs* of one of the chain elements is the *lhs* of the other then there two can potentially be chained together. The requirement for both chain elements to be chained together is that there is a theorem/lemma (**chaining scheme**) in the context that takes the proofs of both chain elements an forms a new statement. For example when having the two chain elements *a [[proof >> b* and *b [[proof0 >> c* (where *proof* states that *R a b* and *proof0* states that *R0 b c* where *R* and *R0* are both binary relations) then those two can be chained together in case there exists a lemma that states that $R\ a\ b \to R0\ b\ c \to R1\ a\ c$, where *R1* is a binary relation). In case the system is not able to automatically find an appropriate lemma/theorem for chaining two chain elements together even though there exists one then the user can use an alternative version of the chain element syntax to specify which chaining lemma the system is supposed to use. To do so one writes *lhs [[lemma]] proof >> rhs* where *lemma* is the chaining lemma. This syntax can also be used in case one wishes to be more explicit.
The heterogeneous chained (pseudo) transitivity proofs also allow the user to specify which target relation is used in a chain element (see fig. 65). This will support the MMT-system in finding an appropriate chaining lemma and also helps with being more clear what a chain element is supposed to represent. To tell the system what the local target relation is one has to use the alternative syntax *lhs [[proof (| target_relation >> rhs* where 'target_relation' is the specified local target relation (see example 64 for a concrete example). This again can also solemnly be used for the purpose of being more explicit when writing the proof. There is also a syntax that lets the user specify the local target relation and and the chaining lemma. To do so one has to use the syntax: *lhs [[chaining_lemma]] proof (| local_target_relation >> rhs.*

4.3 Examples

Example (fig. 66); lines 62 to 69 This example uses the MMT-only implementation but instead of instantiating the MMT-only (homogeneous chained transitivity proofs) theory this example is done within the original theory that defines the MMT-only implementation. That way it is also shown how to define lemmas in a more general way which will be made available

once the user instantiates the theory that contains the MMT-only implementation (even though this example lemma would be quite useless). Further more, using this kind of example is more straight forward since it circumvents the whole instantiation process.
The actual proof states for the values *va*, *va0* (of a type *u*) holds *eq va va0* when *eq va va0*(named vaeqva0) holds(where eq is a transitive and reflexive binary relation on type *u*). To proof this statement only one chain element is needed: one that states that *eq va va0*. As described in the chapter about homogeneous chained transitivity proofs, a chain element is build by combining an **lhs** with and **rhs** and an **connecting proof**. Since the proof that *eq va va0* is already in the context one can simply proof the statement as a single element chain by using *va* as **lhs**, *va0* as **rhs** and *vaeqva0* as **combining proof**.

Example (fig. 66); lines 62 to 69 This example is a simple extension to the first example by forcing a second chain element. This time the goal is to proof that *eq va va1*. Unlike the first proof there is not a theorem in the current context that states just that. But since the **target relation** is transitive and there are two proofs that *eq va va0* and *eq va0 va1* it is possible to generate a transitive proof chain containing booth proofs. To do so two chain elements are needed. The first one containing *va* as **lhs**, *va0* as **rhs** and *vaeqva0* as **combining proof**, the second one containing *va0* as **lhs**, *va1* as **rhs** and *va0eqva1* as **combining proof**. The linking element is *va0* resulting in the proof *va << vaeqva0 >> va0 << va0eqva1 >> va1*.

Example (fig. 67) This example uses an instantiation of the MMT-only theory to proof that two natural number are equal (where “natural numbers” is just a name for another type used to instantiate the MMT-only implementation of the chained transitivity proofs).
To instantiate the MMT-only theory one includes the theory displayed in figure 66 with appropriate parameters (line 78 in fig. 67). The first parameter (*nat*) specifies the carrier type of the **target relation**, the second argument (*nateq*) specifies the **target relation**, the third argument (*refl*) give a proof that the **target relation** is reflexive and the last argument (*nattrans*) provides a proof that the **target relation** (*nateq*) is transitive.
Now that the MMT-only-implementation of the chained transitive proofs has been initialized and included one can proceed to proof the desired theorem.

The goal of this example is to proof that *v* equals *v0* (*nateq v v0*). Since the proof that *nateq v v0* is already in the context, this proof can be implemented using just a single chain element. Basically , the solution is just a translation of the theorem that states that *nateq v v0* into the chained transitivity proof syntax. Again, the chain element consists of the **lhs**, **rhs** and the **combining proof**. The combining proof in this case is the theorem *veqv0*, the **lhs** is *v* and the **rhs** is the v0. The final proof is displayed in line 89 of figure 67.

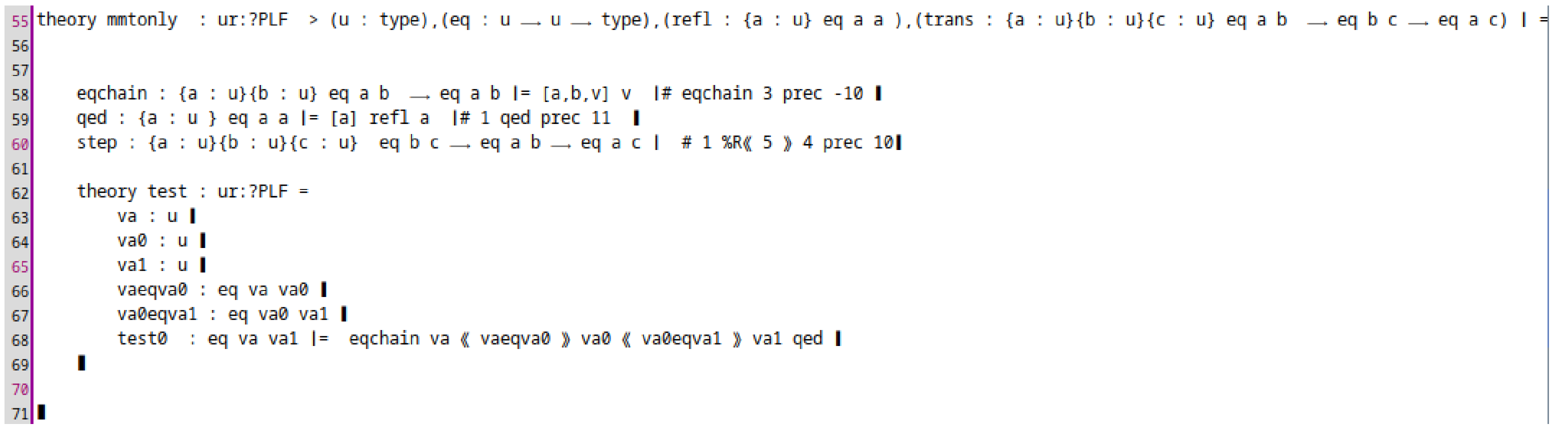
Example (fig. 67); lines 62 to 69 This example is an extension to the previous example. Now the goal is to proof that *nateq v v2*. This time it is not possible to state the proof as a single chain element. This time three elements have to be chained the together incorporating the three proofs in lines 86 to 88. Again the way to construct the chain elements, when there is already a proof stating what the chain element is supposed to represent (like *nateq v v0*), is to just translate it syntactically. So *veqv0* translates to *v << veqv0 >> v0*, *v0eqv1* translates to *v0 << v0eqv1 >> v1* and v1eqv2 translates to *v1 << v0eqv1 >> v2*. Now when combining the single chain elements one looks at what the lhs of the goal (*nateq v v2*) is and searches for the chain element which **lhs** matches the goal-lhs. In this example this is the chain element using the combining proof *veqv0*. Then one looks for the next chain element that has the appropriate **lhs** that matches the **rhs** of the first chain element. The proof then becomes *v << veqv0 >> v0 << v0eqv1 >> v1* where *v0* become the linking element that chains the first two chain elements together. Since this only proofs that *nateq v v1* it is necessary to add the last element to this proof (the chain element using *v1eqv2*). The final proof is displayed in line 90 of figure 67.

Example fig. 69 This example demonstrates the usage of the Scala-rule-implementation of the chained transitivity proofs which slightly deviates from the MMT-only implementation. It also demonstrates other ways how one can supply to the combining proof to a chain element. Further more this example shows how one would normally go about more “naturally” constructing these kinds of proofs instead of just figuring out how to stick together already proven facts as it was done in the previous examples.
To be able to use the Scala implementation it is necessary to instantiate the rewrite-tactic parametric rule as described in the chapter about rewriting

with at least the carrier, a target relation and a proof that this relation is transitive . Otherwise MMT will complain.
The goal is to proof that $a + b + c = c + b + a$. Now the idea is to basically go through the steps required to transform the lhs of the goal to the rhs as one has done many times on paper . The solution and steps now presented are just one of many ways to proof this goal. Also, only *addassoc* and *addcomm* are supposed to be used.
Because c is at the very left of the addition and a is on the very right of the addition the idea is to let them "switch positions". Unfortunately this is not directly possible. Hence one has to apply associativity and commutativity in a way that achieves just that, but in a multi-step way.
The first (chosen) step is to use associativity of plus to transform $a + b + c$ (which is implicitly bracketed as $((a + b) + c)$) to $a + (b + c)$. Since there is no proof that $((a + b) + c) = a + (b + c)$ it has to be constructed/proven. The combining proof can be anything that has the appropriate type. Therefore it is also possible to supply a tactics proof that shows just what is needed. In this particular case one can use the rewrite-tactic and the associativity theorem to do the transformation. The combining proof then reads as *rewrite addassoc a b c* applying the values *a, b, c* to the *addassoc*-lemma. This results in the first step: *a + b + c =<< rewrite addassoc a b c >>= a + (b + c)* (note the slightly changed syntax of the brackets =<< and >>= instead of << and >>, indicating that the Scala implementation of the chained transitivity proofs is being used here).
Next step is to rewrite using commutativity (line 52). Again, the proof is supplied as a tactics script. The proof so far is *a + b + c =<< rewrite addassoc a b c >>= a + (b + c) =<< rewrite addcomm b c >>= a + (c + b)* . The remaining proof is done in a similar fashion as depicted in the lines 54 to 57. A shorter proof that proofs the same is given in 70.
Since writing all the detail of a chained transitivity proof can be quite laborious or distract from the actual important path by cluttering what is actually important, it is possible to leave parts of the chain implicit as shown in figure 71. When parts of the proof are left out like this (under to hood creating unknowns), MMT tries to solve them on its own. That way the user can spare the reader of that proof from unnecessary clutter. Further more this also allows the writer of the to easily modify the proof since it is not necessary to rewrite the parts of the proof that MMT automatically solves. Figure 72 shows a proof that leaves even more parts out including **lhs** and **rhs** at some places.

A good rule of thumb is to either specify the **combining proof** at least partially and therefore to be able to leave the **lhs** and **rhs** unspecified, or to specify the **lhs** and **rhs** and leave the **combining proof** for MMT to figure out.
As one might have noticed the tactic proofs are missing the reflexivity in the end. This is a convenience feature because equality proofs usually end with reflexivity and therefore the chained transitivity reasoning automatically appends them to a proof script if necessary. They can, however, be explicitly stated if wanted (see figure 73). As figure 73 shows, it is also possible to supply multi step proof scripts (again, the rule is just that the supplied term has to be of type ***lhs*** = ***rhs***) of the chain element it belongs to.

heterogeneous example (fig. 64) Figure displays a proof that uses heterogeneous chained transitivity proofs (lines 114 to 117). The proofs are all the same except for the parts that are left implicit. In general, the heterogeneous proofs follow the same structure as the homogeneous. The main difference is that, due to the usage of different local target relations, that one has to be at times more specific and help the system by supplying additional information. In general, the proof engineer can add two more parameters to the chain element: the first is the chaining scheme and the second being the explicit naming of the local target relation. The proof in line 114 leaves the local target relations implicit. The proof in the next line adds the explicit naming of the local target relation in the chain elements. The proofs in the next two lines leaves some parameters explicit but are essentially the same as the ones before. The *fact*-chaining scheme just states that this chain element does not chain with a previous chain or chain element. The next chain element states that $b = c$ which is proven by *h0*. The chaining scheme *nateqtrans* states that the previous chain (i.e. **inner chain**) has to be of type $a = b$ where a and b are arbitrary expressions of type nat and the current chain element has to be of type $a = b$ as well. The chaining scheme also states that the inner chain (which the current chain element will then belong to) will be of type $a = b$ again.
For a change the next two chain elements have two different chaining schemes (*natletrans* and *natlttrans*), one requiring the current chain element be of type $a \leq b$ and resulting in $a \leq d$, and the other that the current chain element is of type $a < b$ which will result in the desired type $a < e$.



```
theory mmtonly  : ur:?PLF  > (u : type),(eq : u ⟶ u ⟶ type),(refl : {a : u} eq a a ),(trans : {a : u}{b : u}{c : u} eq a b  ⟶ eq b c ⟶ eq a c) ❙ =


   eqchain : {a : u}{b : u} eq a b  ⟶ eq a b ❘= [a,b,v] v  ❘# eqchain 3 prec -10 ❙
   qed : {a : u } eq a a ❘= [a] refl a  ❘# 1 qed prec 11  ❙
   step : {a : u}{b : u}{c : u}  eq b c ⟶ eq a b ⟶ eq a c ❘  # 1 %R《 5 》 4 prec 10❙

   theory test : ur:?PLF =
      va : u ❙
      va0 : u ❙
      va1 : u ❙
      vaeqva0 : eq va va0 ❙
      va0eqva1 : eq va0 va1 ❙
      test0  : eq va va1 ❘=  eqchain va 《 vaeqva0 》 va0 《 va0eqva1 》 va1 qed ❙
   ❚

❚
```

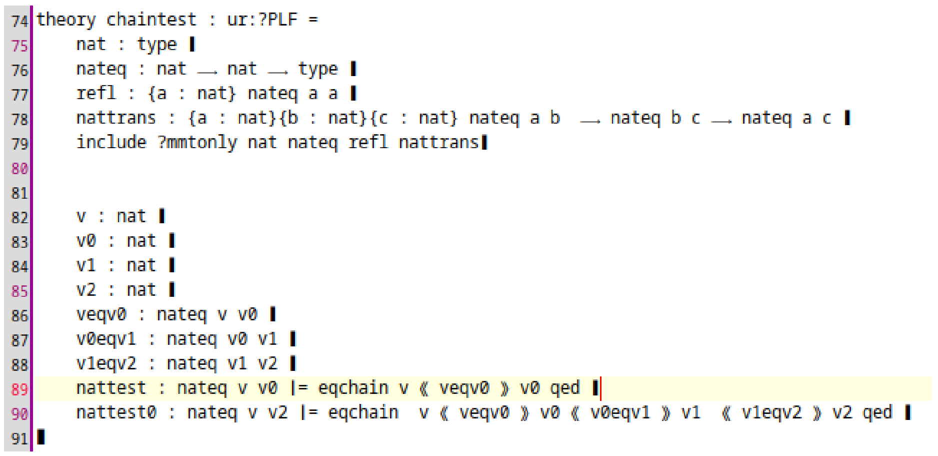
Figure 66: MMT-only implementation of the equality chain reasoning

4.4 Implementation

The implementation can be subdivided into three smaller parts: The MMT only implementation which uses only mechanisms accessible from the MMT environment and then the Scala implementations which again can be separated into two different parts for homogeneous transitivity (i.e. $R\ a\ b \rightarrow R\ b\ c \rightarrow R\ a\ c$ where R is a binary relation) and heterogeneous transitivity (i.e. $R\ a\ b \rightarrow R0\ b\ c \rightarrow R1\ a\ c$ where R, R0, R1 are all potentially different binary relations).

4.4.1 MMT-only implementation

The MMT only implementation basically translates the Agda-implementation of chained equality reasoning to MMT. Unlike the implementations which use the MMT-rule mechanism, this implementation of the chained equality reasoning needs multiple constructs to achieve the same functionality as the other, single relation Scala implementation. The implementation resides within a parameterized theory so it can be instantiated for a concrete type and transitive-reflexive operator (see fig. 66).
The MMT-only implementation consists of three constructs all defined in MMT itself without the need of defining inference and/or typing rules. A chain element is basically implemented as an alternative syntax for applying arguments to the supplied transitivity property. Due to the fact that the first element in the transitivity chain has no inner chain a trick is used to



```
theory chaintest : ur:?PLF =
    nat : type ❙
    nateq : nat ⟶ nat ⟶ type ❙
    refl : {a : nat} nateq a a ❙
    nattrans : {a : nat}{b : nat}{c : nat} nateq a b  ⟶ nateq b c ⟶ nateq a c ❙
    include ?mmtonly nat nateq refl nattrans❙


    v : nat ❙
    v0 : nat ❙
    v1 : nat ❙
    v2 : nat ❙
    veqv0 : nateq v v0 ❙
    v0eqv1 : nateq v0 v1 ❙
    v1eqv2 : nateq v1 v2 ❙
    nattest : nateq v v0 |= eqchain v ⟪ veqv0 ⟫ v0 qed ❙
    nattest0 : nateq v v2 |= eqchain  v ⟪ veqv0 ⟫ v0 ⟪ v0eqv1 ⟫ v1  ⟪ v1eqv2 ⟫ v2 qed ❙
❚
```

Figure 67: Equality chain reasoning using the MMT-only implementation

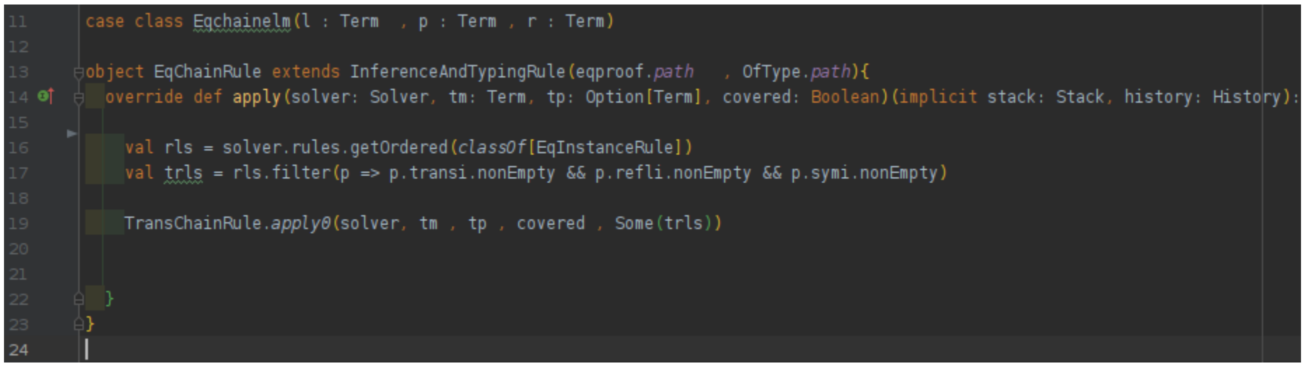
give it an inner chain (generating a reflexivity proof for that gets passed to the first, i.e. rightmost element). The second construct is the *qed*-construct which takes the rightmost element in the chain and returns a trivial proof of reflexivity (line 59). The third construct (*begin*) is just syntax and adds no actual functionality.

4.4.2 Single relation Scala implementation

The single relation Scala implementation uses an inference and typing rule to implement the mechanisms of the chained equality reasoning (or to be more specific the mechanisms of the single chain elements; see figure 68). When the Scala implementation is used, the system checks whether the target relation is transitive. This is done by searching the rule set of the solver for rules of type EqInstanceRule (line 16) and filters those which are transitive and reflexive. After that the system tries to infer or check the type of the current chain element.

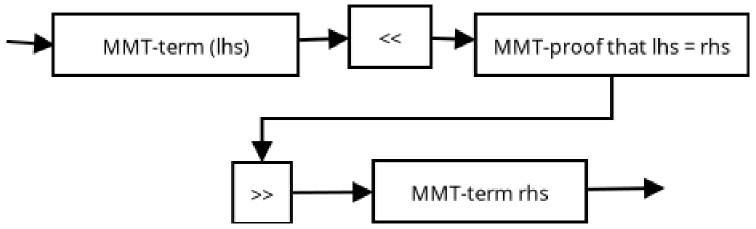
4.4.3 Multi-relation Scala implementation

The multi-relation implementation is done mainly in Scala. Like the single relation implementation it relies on MMT's rule mechanism (i.e. the InferenceAndTypingRules). It does however not require the rewrite tactic instantiation. In general it works much the same as the Scala implementation for the homogeneous transitivity proofs.



```
case class Eqchainelm(l : Term  , p : Term , r : Term)

object EqChainRule extends InferenceAndTypingRule(eqproof.path   , OfType.path){
  override def apply(solver: Solver, tm: Term, tp: Option[Term], covered: Boolean)(implicit stack: Stack, history: History):

    val rls = solver.rules.getOrdered(classOf[EqInstanceRule])
    val trls = rls.filter(p => p.transi.nonEmpty && p.refli.nonEmpty && p.symi.nonEmpty)

    TransChainRule.apply0(solver, tm , tp , covered , Some(trls))


  }
}
```

Figure 68: Inference and typing rule that implements the mechanisms of the chain elements



Due to time constraints it was not possible to implement the heterogeneous chained transitivity proofs (see future work).

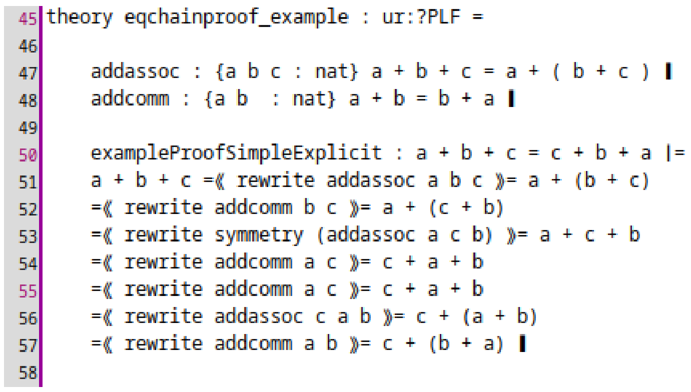
5 Tactics for equality reasoning

5.1 Terminology

- target relation, relation used for rewriting
- Parametric relation: A parametric relation is something of type { $a : A$, $a_1 : A_1$, ... $a_n : A_n$ } **target_relation**

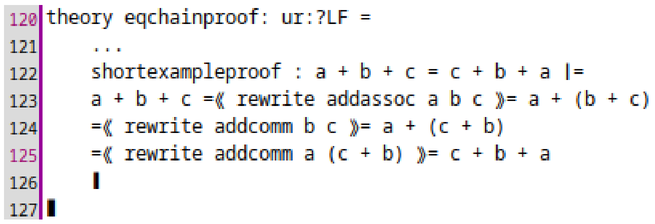
5.2 Introduction

Many proofs involve some kind of equality reasoning. This kind of reasoning can be split into four different subcategories: Reflexivity proofs, symmetry proofs, transitivity proofs, and congruence proofs.
In case one wants to use the tactics one has to include the parametric rule



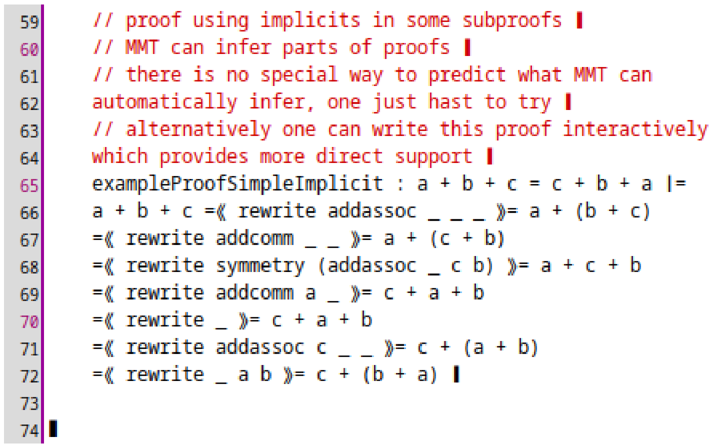
```
theory eqchainproof_example : ur:?PLF =

    addassoc : {a b c : nat} a + b + c = a + ( b + c ) ❙
    addcomm : {a b  : nat} a + b = b + a ❙

    exampleProofSimpleExplicit : a + b + c = c + b + a |=
    a + b + c =⟪ rewrite addassoc a b c ⟫= a + (b + c)
    =⟪ rewrite addcomm b c ⟫= a + (c + b)
    =⟪ rewrite symmetry (addassoc a c b) ⟫= a + c + b
    =⟪ rewrite addcomm a c ⟫= c + a + b
    =⟪ rewrite addcomm a c ⟫= c + a + b
    =⟪ rewrite addassoc c a b ⟫= c + (a + b)
    =⟪ rewrite addcomm a b ⟫= c + (b + a) ❙

```

Figure 69: Example: simple commutativity-proof with three variables



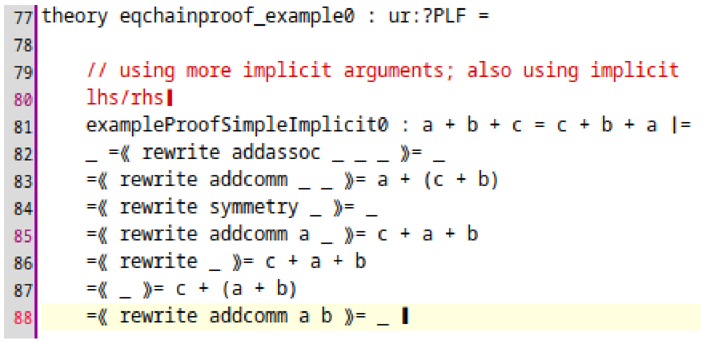
```
theory eqchainproof: ur:?LF =
    ...
    shortexampleproof : a + b + c = c + b + a |=
    a + b + c =⟪ rewrite addassoc a b c ⟫= a + (b + c)
    =⟪ rewrite addcomm b c ⟫= a + (c + b)
    =⟪ rewrite addcomm a (c + b) ⟫= c + b + a
    ❙
❚
```

Figure 70: Example: short version of the proof in figure 69



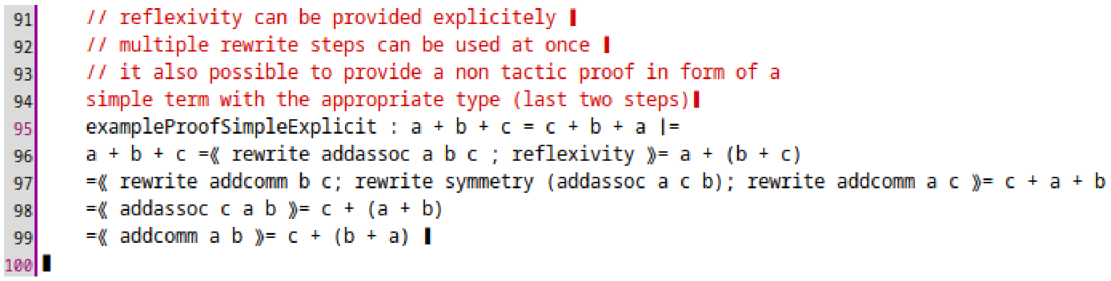
```
    // proof using implicits in some subproofs ❙
    // MMT can infer parts of proofs ❙
    // there is no special way to predict what MMT can
    automatically infer, one just hast to try ❙
    // alternatively one can write this proof interactively
    which provides more direct support ❙
    exampleProofSimpleImplicit : a + b + c = c + b + a |=
    a + b + c =⟪ rewrite addassoc _ _ _ ⟫= a + (b + c)
    =⟪ rewrite addcomm _ _ ⟫= a + (c + b)
    =⟪ rewrite symmetry (addassoc _ c b) ⟫= a + c + b
    =⟪ rewrite addcomm a _ ⟫= c + a + b
    =⟪ rewrite _ ⟫= c + a + b
    =⟪ rewrite addassoc c _ _ ⟫= c + (a + b)
    =⟪ rewrite _ a b ⟫= c + (b + a) ❙

❚
```

Figure 71: Same proof as in 69 but with holes left for MMT to figure out



```
theory eqchainproof_example0 : ur:?PLF =

   // using more implicit arguments; also using implicit
   lhs/rhs❙
   exampleProofSimpleImplicit0 : a + b + c = c + b + a |=
   _ =⟪ rewrite addassoc _ _ _ ⟫= _
   =⟪ rewrite addcomm _ _ ⟫= a + (c + b)
   =⟪ rewrite symmetry _ ⟫= _
   =⟪ rewrite addcomm a _ ⟫= c + a + b
   =⟪ rewrite _ ⟫= c + a + b
   =⟪ _ ⟫= c + (a + b)
   =⟪ rewrite addcomm a b ⟫= _ ❙
```

Figure 72: Same proof as in 71 but with even more implicites left for MMT to figure out



```
   // reflexivity can be provided explicitely ❙
   // multiple rewrite steps can be used at once ❙
   // it also possible to provide a non tactic proof in form of a
   simple term with the appropriate type (last two steps)❙
   exampleProofSimpleExplicit : a + b + c = c + b + a |=
   a + b + c =⟪ rewrite addassoc a b c ; reflexivity ⟫= a + (b + c)
   =⟪ rewrite addcomm b c; rewrite symmetry (addassoc a c b); rewrite addcomm a c ⟫= c + a + b
   =⟪ addassoc c a b ⟫= c + (a + b)
   =⟪ addcomm a b ⟫= c + (b + a) ❙
❚
```

Figure 73: Same proof as in figure 72 but using multistep proof scripts, also stating the final reflexivity tactic explicitly

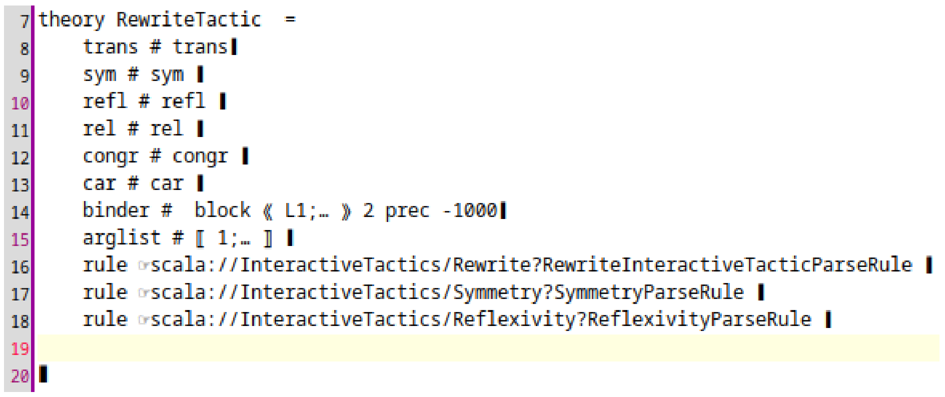
parameter	parameter name	expected type	tactic enabled	mandatory
carrier	car	type	-	yes
relation	rel	car → car → type	-	yes
symmetry	sym	{a b : car} rel a b → rel b a	symmetry	no
transitivity	trans	{a b c : car} rel a b → rel b c → rel a c	rewrite	no
congruence	cong	{a b c : car} rel a b → rel b c → rel a c	rewrite	no
reflexivity	refl	{a : car} rel a a	reflexivity	no

Figure 74: list of parameters for the EqInstancRule

as shown schematically in fig. 76. This rule can be instantiated with various parameter which will enable different tactics, depending on the passed parameters. Tableau 74 shows what parameters can or must be passed to the rule. **relation** specifies the targeted binary relation. **car** specifies the carrier for the target homomorph relation. **car** and **rel** have to be specified. **symmetry**, **reflexivity**, **transitivity** and **congruence** specify the equality properties for the target relation. Each parameter is specified by first stating its name and then followed by the actual parameter (names are defined in the theory in fig. 75). Therefore it is possible to change the order in which the parameters are passed, sort of like named parameters in programming languages like Scala. The parameters can also be parameterized over several variables. This works kind like a lambda/pi binder which specifies the names and types ahead of the actual parameters passed to the rule. Those bound names can also depend on each other much the same way they can in a lambda/pi binder.

Example for rewrite rule instantiation (fig. 78) To enable the set of tactics for equality reasoning one first has to instantiate the *EqGenerateRule* as shown in line 12 and 13. The rule works by passing named parameters. The parameters available are:

- car: Carrier
- rel: Target relation; a binary relation that is of type *car* → *car* → *car* → *type*
- refl: A property that shows that **rel** is reflexive (type: a: car rel a a)



```
theory RewriteTactic  =
    trans # trans❙
    sym # sym ❙
    refl # refl ❙
    rel # rel ❙
    congr # congr ❙
    car # car ❙
    binder #  block 《 L1;… 》 2 prec -1000❙
    arglist # ⟦ 1;… ⟧ ❙
    rule ☞scala://InteractiveTactics/Rewrite?RewriteInteractiveTacticParseRule ❙
    rule ☞scala://InteractiveTactics/Symmetry?SymmetryParseRule ❙
    rule ☞scala://InteractiveTactics/Reflexivity?ReflexivityParseRule ❙

❚
```

Figure 75: Theory defining the parametric equality-rule

- sym: A property that shows that **res** is symmetric (type: a b : car (rel a b → rel b a) ∧ (rel b a → rel a b))
- trans: A property that shows that **res** is transitive (type: a b c : car rel a b → rel b c → rel a c)

One has to at least pass the **car** and **rel** parameter. Depending on what other parameters have been passed the *reflexivity*,*symmetry* and/or *rewrite*-tactic are enabled for the specified **rel** (a list of what each parameter means and what tactic it enables is listed in figure 74). It is also possible to parameterize over the passed parameters. For that reason the *EqInstanceRule* offers to put parameters between the << and >>. This allows for parameters to be (partially) generic instead of being fixed. In the example, the parameter *A : type* is added in front of the passed parameters (car, rel, etc).

5.3 reflexivity

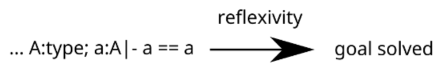


Figure 79: semi-formal description of the reflexivity tactic

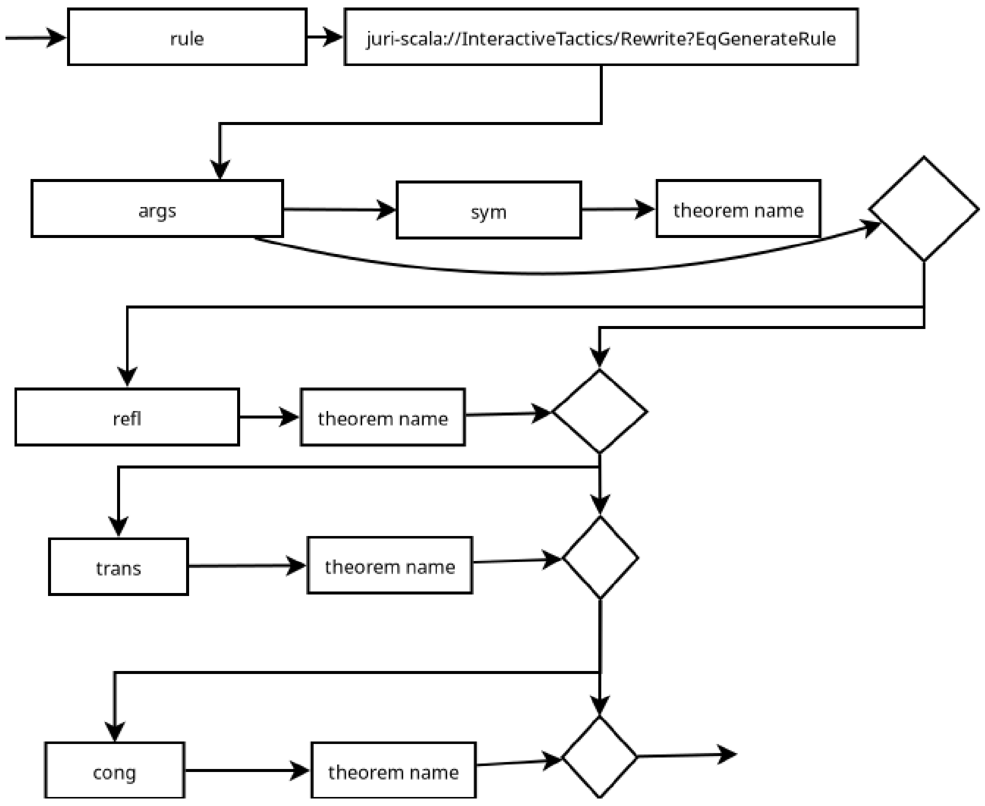


Figure 76: Syntax for the rewrite-rule

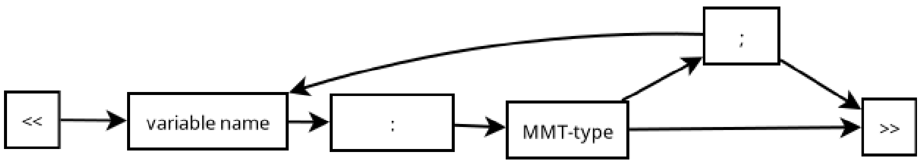
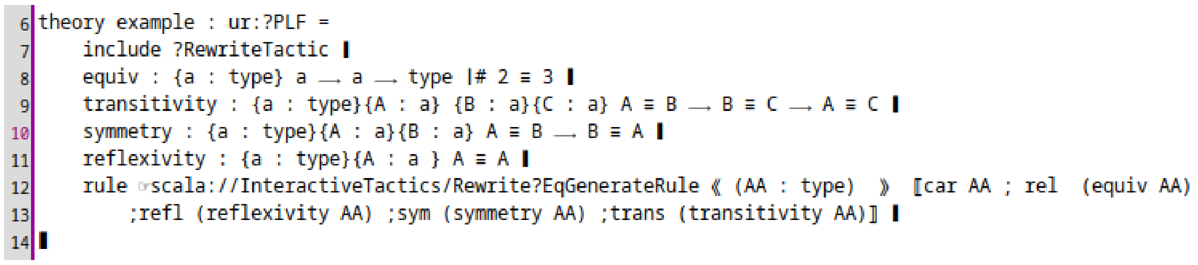


Figure 77: args-syntax used in fig. 76



```
theory example : ur:?PLF =
    include ?RewriteTactic ❙
    equiv : {a : type} a ⟶ a ⟶ type |# 2 ≡ 3 ❙
    transitivity : {a : type}{A : a} {B : a}{C : a} A ≡ B ⟶ B ≡ C ⟶ A ≡ C ❙
    symmetry : {a : type}{A : a}{B : a} A ≡ B ⟶ B ≡ A ❙
    reflexivity : {a : type}{A : a } A ≡ A ❙
    rule ☞scala://InteractiveTactics/Rewrite?EqGenerateRule ❬ (AA : type)  ❭ ⟦car AA ; rel  (equiv AA)
        ;refl (reflexivity AA) ;sym (symmetry AA) ;trans (transitivity AA)⟧ ❙
❚
```

Figure 78: Example instantiation

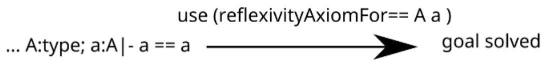


Figure 80: a reflexivity step without the reflexivity tactic; *reflexivityAxiomFor==* is just a made up name that states that $\{A : type\}\{a : A\}\ a == a$

These kinds of proofs occur when the goal is of form $a = a$. The reflexivity tactic would then solve the goal. Introducing a tactic for these kinds of proofs isn't strictly necessary (as most tactics are, except for a few fundamental tactics) but convenient. The advantage of the reflexivity tactic is that it automatically deals with some of the more fundamental details like which reflexivity axiom/theorem to use (depending on which relation is used) and the arguments applied to that axiom/theorem. For example is the reflexivity for equals "=" different for less then or equal (<=). The figures 79 and 80 show the difference between using the reflexivity-tactic and doing it without it.

5.4 symmetry

The symmetry tactics comes into play when a term of the form **lhs = rhs** should be flipped to **rhs = lhs**. This tactics can be applied to either the goal, then called without any further arguments, to a hypothesis in the local context, then passing the name of the hypothesis to the symmetry tactic. When applied to a hypothesis, symmetry will replace the old hypothesis with a new one where lhs and rhs are flipped, keeping the old name for the hypothesis (see figure 81 for a formal description and examples).

5.5 rewrite

5.5.1 Motivation

Doing "rewriting" by using theorems directly without any special support by an interactive system can become very quickly unfeasible. This is mostly due to the user having to keep track of lots of information in a very confuse looking proof.
The figures 82 and 83 show the stark difference between a proof done using tactics and one using a plain lambda term to proof $a + b + c + d = d + c + b + a$. At a certain complexity level lambda terms loose their ability to be easier to understand then tactics and that point is usually reached fairly

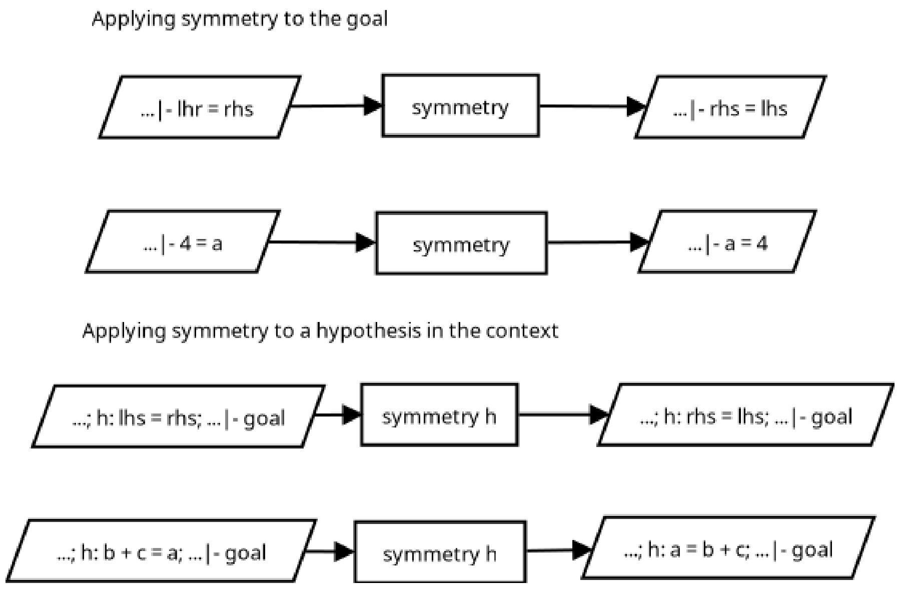
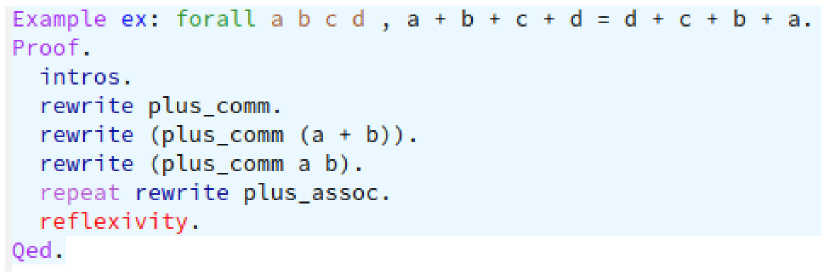
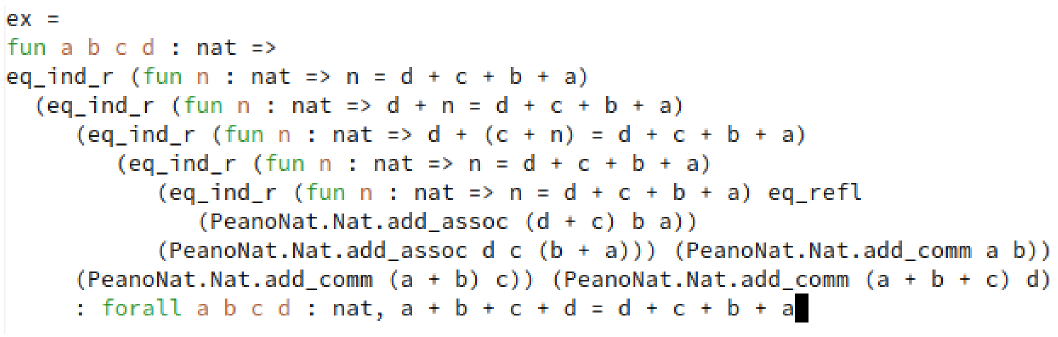

Applying symmetry to the goal
...|- lhr = rhs
symmetry
...|- rhs = lhs
...|- 4 = a
symmetry
...|- a = 4
Applying symmetry to a hypothesis in the context
...; h: lhs = rhs; ...|- goal
symmetry h
...; h: rhs = lhs; ...|- goal
...; h: b + c = a; ...|- goal
symmetry h
...; h: a = b + c; ...|- goal

Figure 81: Description and example of how the symmetry tactic works: The top two lines show how to apply the symmetry tactic to the goal and the bottom two how to apply it to a hypothesis



```
Example ex: forall a b c d , a + b + c + d = d + c + b + a.
Proof.
  intros.
  rewrite plus_comm.
  rewrite (plus_comm (a + b)).
  rewrite (plus_comm a b).
  repeat rewrite plus_assoc.
  reflexivity.
Qed.
```

Figure 82: using tactics



```
ex =
fun a b c d : nat =>
eq_ind_r (fun n : nat => n = d + c + b + a)
  (eq_ind_r (fun n : nat => d + n = d + c + b + a)
     (eq_ind_r (fun n : nat => d + (c + n) = d + c + b + a)
        (eq_ind_r (fun n : nat => n = d + c + b + a)
           (eq_ind_r (fun n : nat => n = d + c + b + a) eq_refl
              (PeanoNat.Nat.add_assoc (d + c) b a))
           (PeanoNat.Nat.add_assoc d c (b + a))) (PeanoNat.Nat.add_comm a b))
     (PeanoNat.Nat.add_comm (a + b) c)) (PeanoNat.Nat.add_comm (a + b + c) d)
     : forall a b c d : nat, a + b + c + d = d + c + b + a
```

Figure 83: using a plaint term for the proof

quickly.

5.5.2 Description

The rewrite tactic replaces low level handling of theorems with a more abstract, high level command that takes care of some of the low level details. It's syntax is given in figure 84.
In general the rewrite tactic takes an argument of the form **eq lhs rhs** (lhs = left hand side; rhs = right hand side; eq = binary transitive relation), called rewrite relation, and replaces occurrences of lhs with rhs in a target term. The tactic always gets called with it's name "rewrite" first. Alternatively it can also be called with an abbreviated "rew". After an MMT-term is specified which states the binary relation and its arguments used for the rewrite, i.e. a term of type **eq lhs rhs**, several additional arguments can be supplied to the rewrite tactic which modify its behavior. In total there are four modifiers: the using-modifier which specifies what transitivity or congruence theorem to use by the rewrite tactic, the in-modifier which signals when specified whether to apply rewriting to a specific hypothesis , the at-modifier which specifies which specific occurrence of lhs in the target term to rewrite and the where-modifier which specifies how to (partially) instantiate the transitivity/congruence theorem (potentially specified by the using-modifier).
Like many tactics that change the proof state in a significant way the rewrite tactic is also represented by a proof term. Therefore it is important to recognize that the rewrite tactic is used for two kinds of rewriting: Transitive rewriting and congruence rewriting. Transitivity rewriting targets terms that

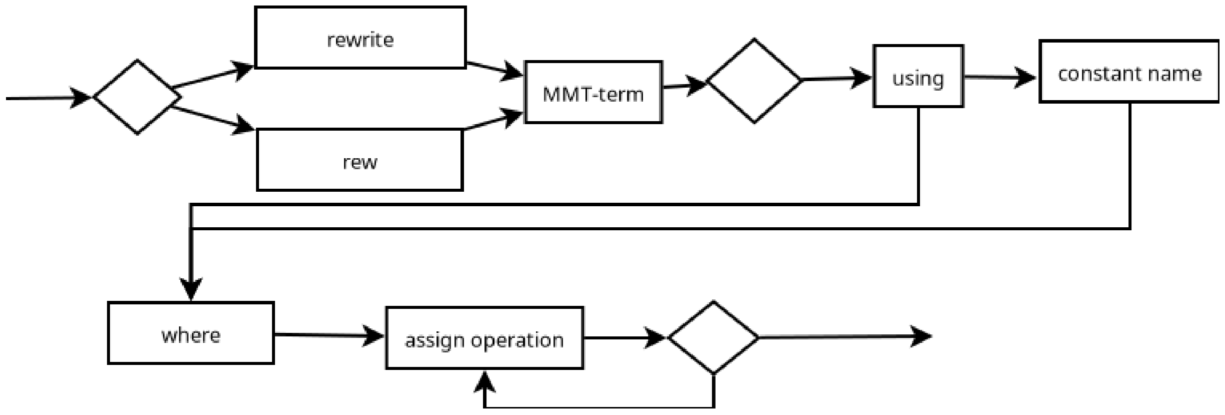


Figure 84: Syntax of the rewrite tactic

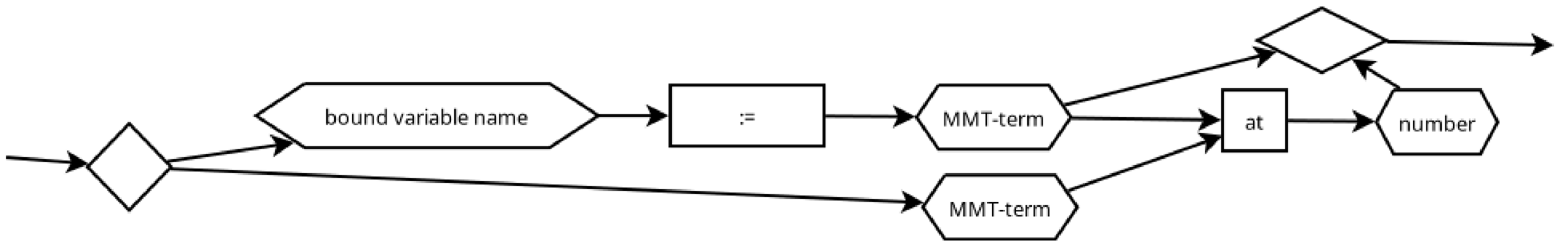


Figure 85: Syntax of the assign operation used in figure 84

can be used in the same transitivity property together with the rewrite relation. In the schematic description in fig. 86, assumes that there is a transitivity property for a (not more precisely specified) equality of type **{A:type, a b c: A} a = b → b = c → a = c** (now called eqtrans). The rewrite tactic uses the hypothesis **h** with type **a = b** to rewrite every occurrence of **a** with **b** in the conclusion. Since the conclusion is also an equality statement the rewrite will be a transitive rewrite. The newly generated goal will be the "mid-part" of the transitivity chain in **a = b → b = c → a = c**. Congruence rewriting on the other hand targets sub-terms in the target term and replaces them with regards to the specified rewriting relation accordingly. Fig. 88 shows the general mechanism of congruence rewriting. The goal **P a** is not an equality like the hypothesis **h**. Therefore using **h** to rewrite the goal to **P b** cannot be a transitive rewrite, instead the congruence property (now called eqcong) is used.

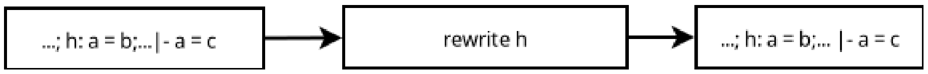


Figure 86: rewrite using transitivity (in case there is exists a transitivity property approximately of type **{A:type, a b c: A} a = b → b = c → a = c**)

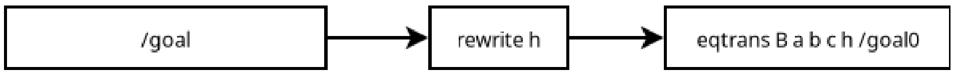


Figure 87: Proof term progression for the rewrite in fig. 86

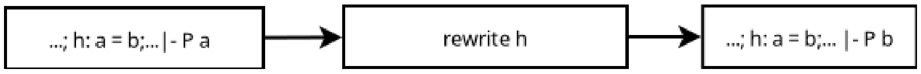


Figure 88: rewrite using congruence in case there exists a congruence property for **P** and equality approximately of type **{A: type, P:A → prop, a b : A } a = b → P b → P a**

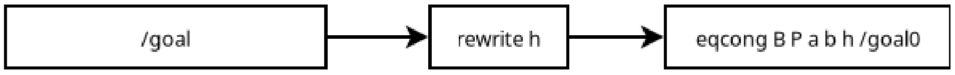


Figure 89: Proof term progression for the rewrite in fig. 88

5.5.3 Examples

Simple abstract example The goal is to proof that *h: c=b; h0: a=c* $\vdash$ $a = b$ (without further specifying which type they are). The steps are to first replace **a** with **c** by using hypothesis h0 and then replace **c** with **b** by using hypothesis h. The rewrite tactic works by taking the lhs of the rewrite relation (in the first rewrite step the rewrite relation is the type of h0, i.e. $a = c$) and replace every occurrence of the lhs in the target term (here the target term is the goal) with the rhs of the rewrite relation.
The graphical depiction of the proof is given in figure 90.
When the rewrite tactic is called it searches for an appropriate rewrite theorem that have been registered via the parametric EqInstace-rule. For this example the example instantiation from 78 . Now when the *rewrite*-tactic is called with *h0* under the hood *transitivity* (the rewrite theorem) is called like this: *transitivity a c b h0 _* (where the underscore is the new hole).

Rewriting in a hypothesis The goal is to proof that *h: b=a; h0: P b* $\vdash$ *P a*(again without further specifying which type they are for simplicities sake). To rewrite in the hypothesis one adds the *in*-parameter at the end of the call to the rewrite tactic. The target term of the rewrite is *h0* using h as rewrite relation.
The graphical depiction of the proof is given in figure 91. Like in the previous example the system will take *transitivity* as rewrite theorem.

Explicitly stating the rewrite theorem Figure 92 shows the example as a whole. The proof is fairly simple. For documentation purposes the rewriting should explicitly state what rewrite theorem should be used (not to be confused with rewriting relation). The first rewrite (*rewrite h0 using eqtrans* where *eqtrans* is the rewrite theorem) just states that it is using a common transitivity theorem which one would expect the rewrite theorem would use anyway. The second one uses a different rewriting theorem, named *eqtrans4*. Both *eqtrans* and *eqtrans4* have to be accessible somewhere in the current (global) proof context when used. Usually the problem with using *eqtrans4* would be that the system usually can't figure out all the forall-bound variables of *eqtrans4* since it does not have enough information (it just knows about the conclusion and one rewrite relation, but in fact it would need two, see the next example for how to pass more information to the *rewrite*-tactic). The special part about using *eqtrans4* here is that since there is no need for an extra transitivity step, the system can figure out all information it needs and generates two goals (one trivial) accordingly. *eqtrans4* gets instantiated as follows: x = c, y = d ,z = b , u = b. To see an example where *eqtrans4* gets used directly see fig. 96.

Explicitly instantiating parts of a rewrite theorem The figures 93, 94 and 95 show the almost same proof using various ways to pass additional parameters to the used rewriting theorem *eqtrans*. The first rewrite in 93 instantiates the named/bound variables in *eqtrans* (type: { x y z } x = y → y = z → x = z) by referencing them by name (*x := a*; *y := c*; *z := b*). This instantiates *eqtrans* to $a = b \rightarrow b = c \rightarrow a = c$.
The second rewrite uses a slightly other way to provide extra arguments. This time the the arguments are provided by indicating at which position they are to be inserted.
Figure 94 shows a variation in which the rewrite theorem is left implicit. In this case one has to know which one the system will choose to be able to appropriately supply additional arguments.
Figure 95 shows that one can also mix both supply styles even for one argument like in the first rewrite where *b* is provided via *z := b at 2* (third additional argument). In cases like these the positional argument takes precedence over the name, but the name of the argument/variable will be checked if it actually matches (i.e. whether the argument at position 2 is actually a named argument and whether the name is actually *z*).
Using explicit arguments one can rework the example from fig. 92 into a more readable version presented in 96.

Rewriting in sub-expressions The general functionality of how the *at*-parameter works is displayed in 97. When the rewrite tactic is called with an *at*-parameter it goes through the term in a depth-first manner, searching for (sub-)expressions that can be rewritten using the rewrite relation passed. When the at parameter is a natural number *n* then the rewrite tactic will take the nth possible sub-expression and rewrites it. When a lambda is passed instead of a natural number the rewrite will take place at the position the lambda parameter marks.(see fig. 99)
The concrete example in 98 replaces in the conclusion the right *P a* that is applied to *Q*.

Mixed example This example (100) uses a rewrite that uses all types of additional arguments (using,at,in,where). Concretely this rewrite means that the rewrite relation $a = c$ named h should be taken to rewrite the hypothesis *h0* (of type *P(a + b + a)*). Also the *eqcong*-lemma ({A }{ P} { x y : A } x = y → P x → P y) is used for rewriting instantiated with a for x. Since only the second a in *P(a + b + a)* the rewrite has an *at* argument that forces the rewrite to only take place at the second a.

Example for transitive rewrite The goal in 101 is to proof a more complex version of the more common transitivity theorem $\mathbf{a = b \to b = c \to a = c}$. To achieve this the simplest strategy is to rewrite each hypothesis **h, h0, h1** one after the other and then apply reflexivity. Since the outermost constant term is the same relation as the rewrite relation the system will pick the transitive property in case one was specified when the EqInstanceRule was specified. In case non was specified the rewrite will fall back to the congruence property. All three rewrites in this example will use the transitivity property if one was specified. The resulting term will be *[h h0 h1] eqtrans h (eqtrans h0 (eqtrans h1 eqrefl))* (eqrefl is the reflexivity property specified during the EqInstanceRule instantiation).

Example congruence rewriting The goal in 102is to proof that if **n** is even so is **m**. Since the hypothesis **h0** states that $\mathbf{m = n}$ this holds true. The only thing that has to be done is to rewrite **m** with **n** in the goal **isEven m**. To do so the rewrite tactic is called with the hypothesis **h0**. The rewrite tactic then rewrites every occurrence of **m** in the goal with **n**, resulting in the new goal **isEven n**. This can then be trivially solved by using the hypothesis **h** using the use-tactic. Since the rewrite in this case can't be expressed as a transitive statement the rewrite tactic takes the congruence property. The resulting term is *eqcongr m n ([x] isEven x) h*.

5.5.4 Implementation

This subchapter is just a rough description of the implementation of the EqInstanceRule and the rewrite tactic itself. For a deeper understanding and technical details it is recommended to read the commented source code and interactively execute it.

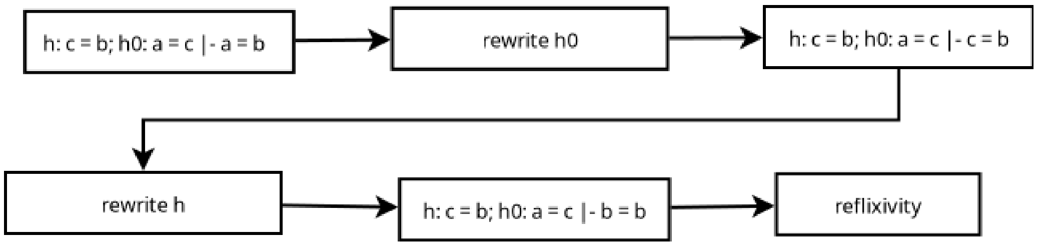

h: c = b; h0: a = c |- a = b
rewrite h0
h: c = b; h0: a = c |- c = b
rewrite h
h: c = b; h0: a = c |- b = b
reflixivity

Figure 90: Simple example using the rewrite tactic

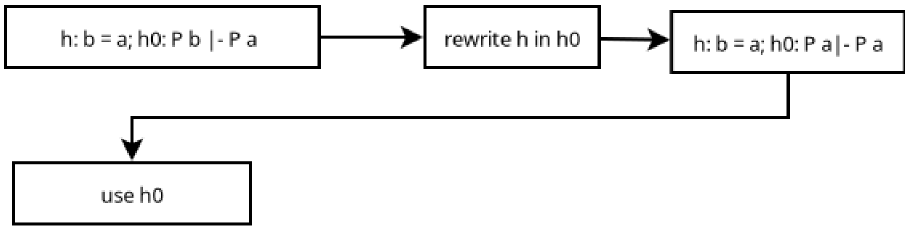


Figure 91: Simple example using the rewrite tactic to rewrite in a hypothesis

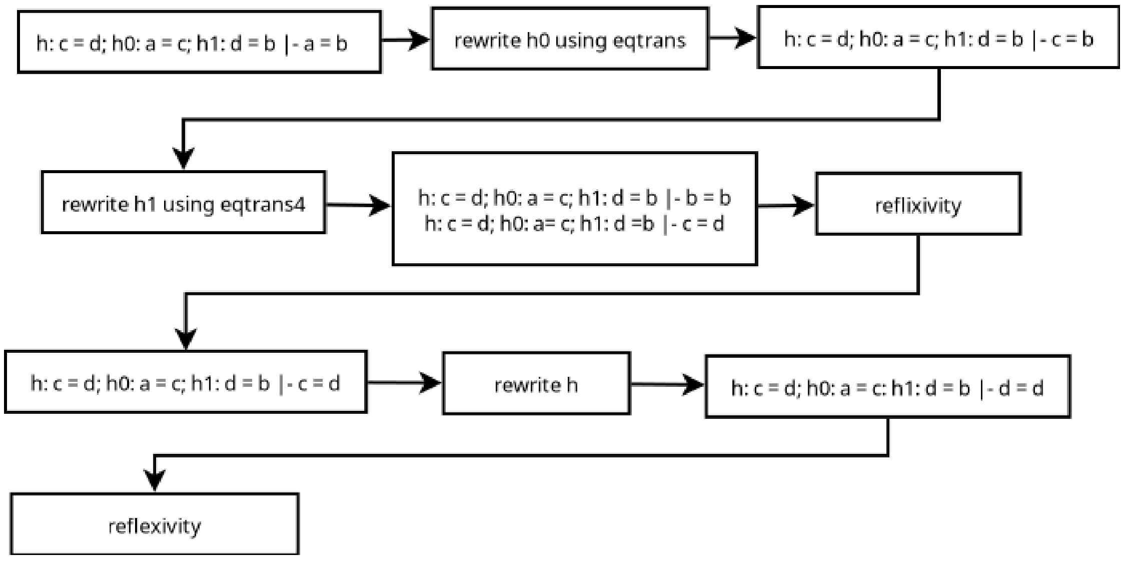
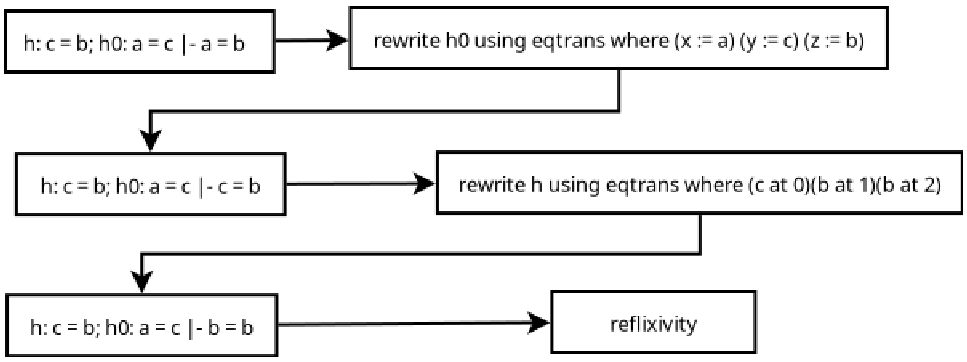
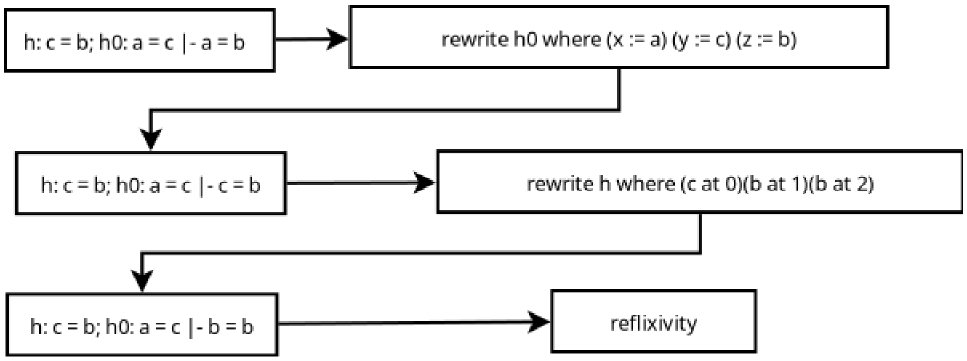
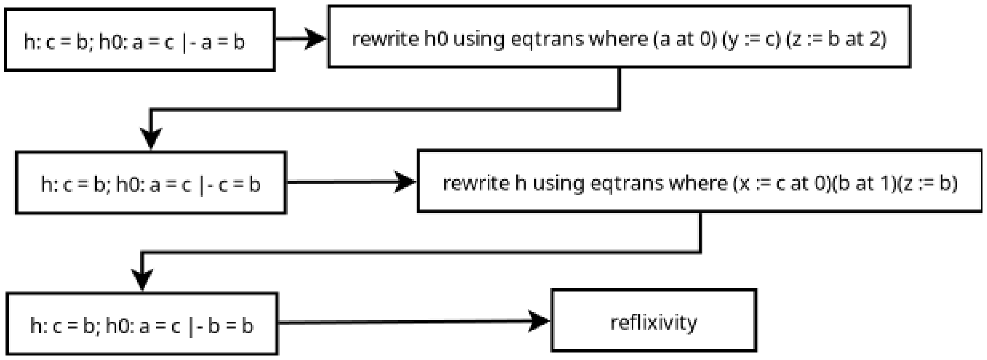

h: c = d; h0: a = c; h1: d = b |- a = b
rewrite h0 using eqtrans
h: c = d; h0: a = c; h1: d = b |- c = b
rewrite h1 using eqtrans4
h: c = d; h0: a = c; h1: d = b |- b = b
h: c = d; h0: a= c; h1: d =b |- c = d
reflixivity
h: c = d; h0: a = c; h1: d = b |- c = d
rewrite h
h: c = d; h0: a = c: h1: d = b |- d = d
reflexivity

Figure 92: Rewrite example where the theorem to use for rewriting is explicitly supplied; *eqtrans* : $\{\ x\ y\ z\ \}\ x = y \rightarrow y = z \rightarrow x = z$; *eqtrans4* : $\{\ x\ y\ z\ u\ \}\ x = y \rightarrow y = z \rightarrow z = u \rightarrow x = u$


h: c = b; h0: a = c |- a = b
rewrite h0 using eqtrans where (x := a) (y := c) (z := b)
h: c = b; h0: a = c |- c = b
rewrite h using eqtrans where (c at 0)(b at 1)(b at 2)
h: c = b; h0: a = c |- b = b
reflixivity
Figure 93: explicitly instantiating the rewrite theorem


h: c = b; h0: a = c |- a = b
rewrite h0 where (x := a) (y := c) (z := b)
h: c = b; h0: a = c |- c = b
rewrite h where (c at 0)(b at 1)(b at 2)
h: c = b; h0: a = c |- b = b
reflixivity
Figure 94: variation of 93


h: c = b; h0: a = c |- a = b
rewrite h0 using eqtrans where (a at 0) (y := c) (z := b at 2)
h: c = b; h0: a = c |- c = b
rewrite h using eqtrans where (x := c at 0)(b at 1)(z := b)
h: c = b; h0: a = c |- b = b
reflixivity
Figure 95: variation of 93

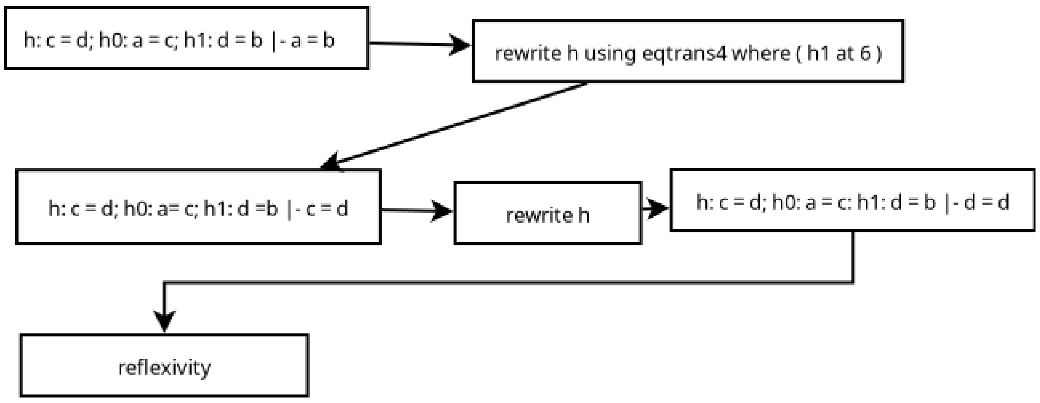

h: c = d; h0: a = c; h1: d = b |- a = b
rewrite h using eqtrans4 where (h1 at 6)
h: c = d; h0: a= c; h1: d =b |- c = d
rewrite h
h: c = d; h0: a = c: h1: d = b |- d = d
reflexivity

Figure 96: the example from 92 but using *eqtrans4* directly

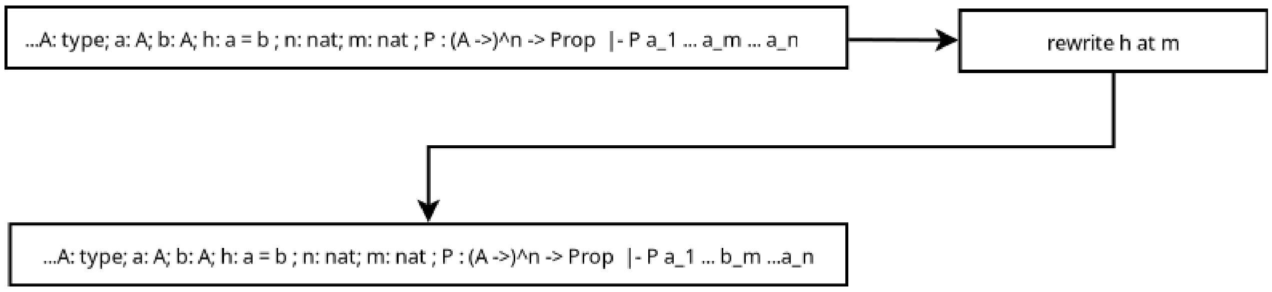

...A: type; a: A; b: A; h: a = b ; n: nat; m: nat ; P : (A ->)^n -> Prop |- P a_1 ... a_m ... a_n
rewrite h at m
...A: type; a: A; b: A; h: a = b ; n: nat; m: nat ; P : (A ->)^n -> Prop |- P a_1 ... b_m ...a_n

Figure 97: Abstract description of the *at*-parameter; the subscript numbers are just for marking the position not for making the parameters actually distinct (i.e. a_1 is equal to a_2 but a_1 is the first parameter for P where as a_2 is the second one)

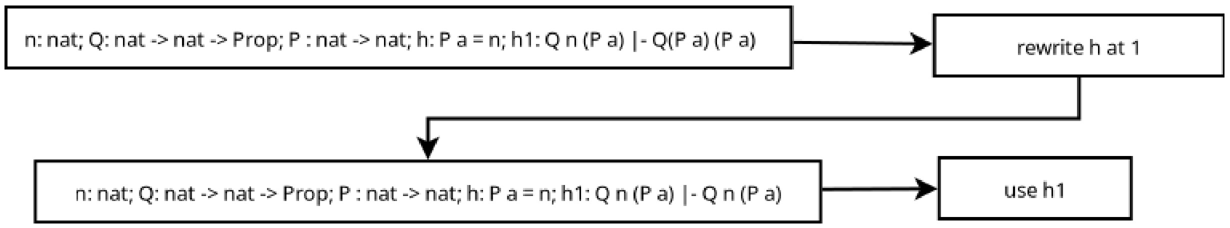

n: nat; Q: nat -> nat -> Prop; P : nat -> nat; h: P a = n; h1: Q n (P a) |- Q(P a) (P a)
rewrite h at 1
n: nat; Q: nat -> nat -> Prop; P : nat -> nat; h: P a = n; h1: Q n (P a) |- Q n (P a)
use h1

Figure 98: rewriting in a subexpression selected by position

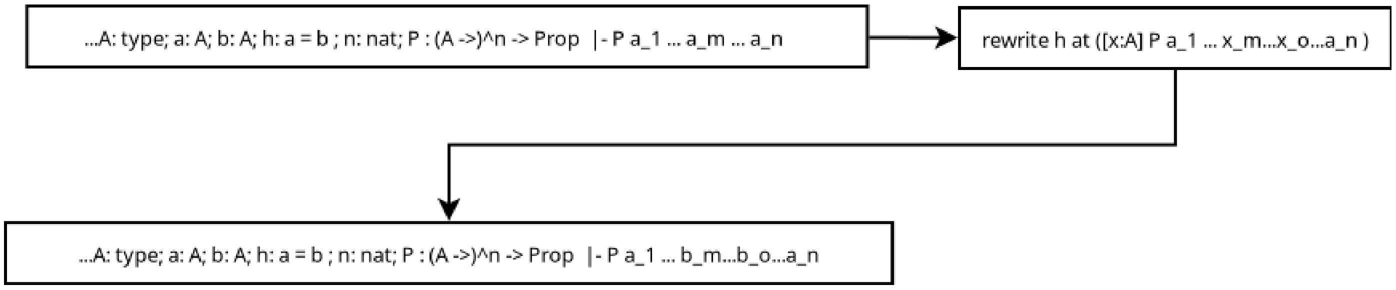

...A: type; a: A; b: A; h: a = b ; n: nat; P : (A ->)^n -> Prop |- P a_1 ... a_m ... a_n
rewrite h at ([x:A] P a_1 ... x_m...x_o...a_n)
...A: type; a: A; b: A; h: a = b ; n: nat; P : (A ->)^n -> Prop |- P a_1 ... b_m...b_o...a_n

Figure 99: rewriting in a subexpression selected using a context

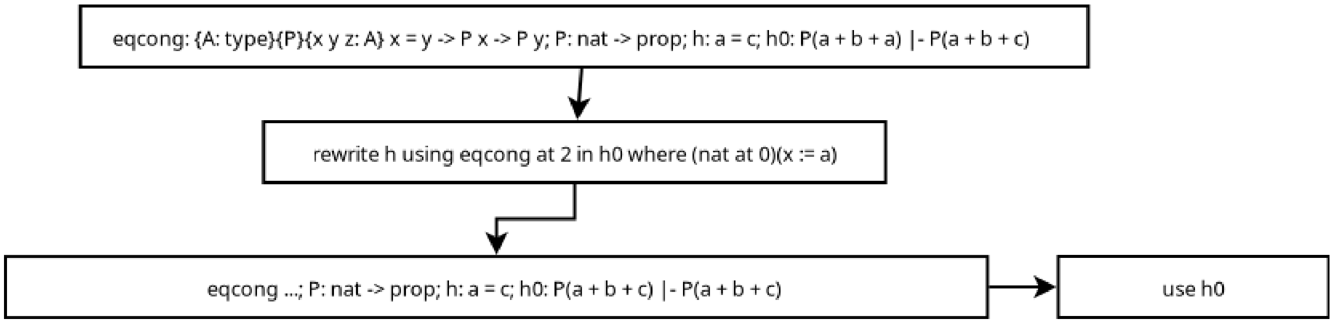

eqcong: {A: type}{P}{x y z: A} x = y -> P x -> P y; P: nat -> prop; h: a = c; h0: P(a + b + a) |- P(a + b + c)
rewrite h using eqcong at 2 in h0 where (nat at 0)(x := a)
eqcong ...; P: nat -> prop; h: a = c; h0: P(a + b + c) |- P(a + b + c)
use h0

Figure 100: An example that uses all the available additional argument types

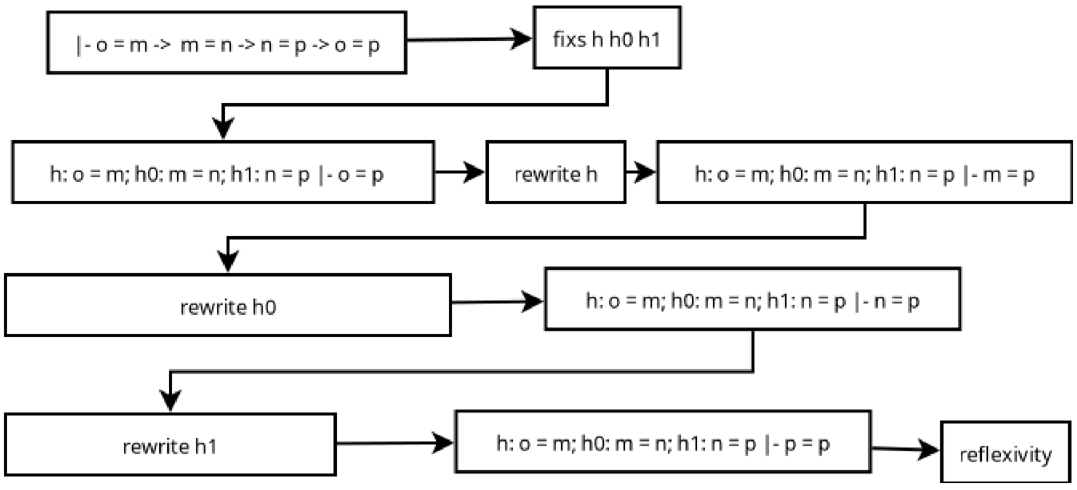

|- o = m -> m = n -> n = p -> o = p
fixs h h0 h1
h: o = m; h0: m = n; h1: n = p |- o = p
rewrite h
h: o = m; h0: m = n; h1: n = p |- m = p
rewrite h0
h: o = m; h0: m = n; h1: n = p |- n = p
rewrite h1
h: o = m; h0: m = n; h1: n = p |- p = p
reflexivity

Figure 101: Example for fig. transitive rewriting

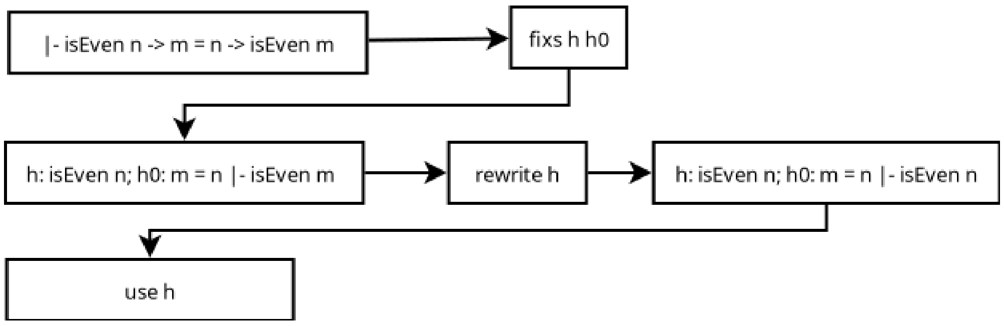
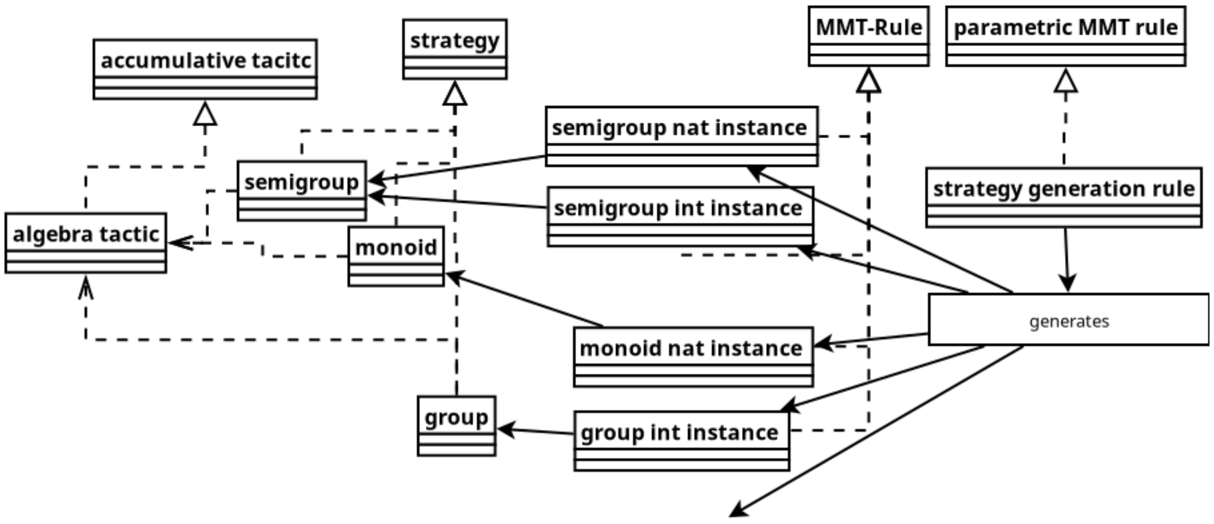

|- isEven n -> m = n -> isEven m
fixs h h0
h: isEven n; h0: m = n |- isEven m
rewrite h
h: isEven n; h0: m = n |- isEven n
use h

Figure 102: Example for fig. congruence rewriting; proving that if n is even so is m if n = m

EqInstanceRule The MMT half of the implementation has already been introduced in 75. Basically the EqInstanceRule takes a lambda which as body has the actual parameters. The "lambda binder" is syntactically represented by <<>> and takes arguments just like the normal lambda binder. The parameters (i.e. the body) are all separated by semi colon and consist of the name of the parameter and its value. The parameters are parameterized over the variables bound by the <<>>.
The rule itself is implemented as an object that derives from the ParametricRule trait. When the rule is called it checks whether the passed parameters have the right type (for example whether the parameter for reflexivity has the type $\{ a \} \; a = a$). Then it generates a new object of the EqInstance class which holds information like what parameters where in the <<>> and which and what parameters where passed, i.e. what properties hold for the passed relation.

Rewrite Tactic The rewrite tactic, like most tactics, consists mainly of its execute-method. When the rewrite tactic's execute-method is called it receives the rewrite relation and the additional parameters. It first checks whether it can either use the transitive property or congruence property with regards to the rewrite relation. It does so by checking the shape of the goal and the rewrite relation. After that it applies the passed *where* arguments, the rewrite relation and the conclusion to the rewrite theorem tries to infer missing parameters and check whether the everything type checks properly. When the *at* parameter is passed and/or the congruence property is used for rewriting the rewrite tactic specifies the P-parameter for the congruence



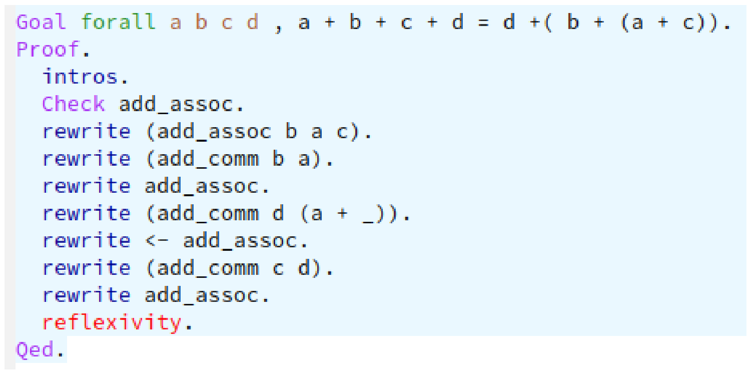

```
Goal forall a b c d , a + b + c + d = d +( b + (a + c)).
Proof.
  intros.
  Check add_assoc.
  rewrite (add_assoc b a c).
  rewrite (add_comm b a).
  rewrite add_assoc.
  rewrite (add_comm d (a + _)).
  rewrite <- add_assoc.
  rewrite (add_comm c d).
  rewrite add_assoc.
  reflexivity.
Qed.
```

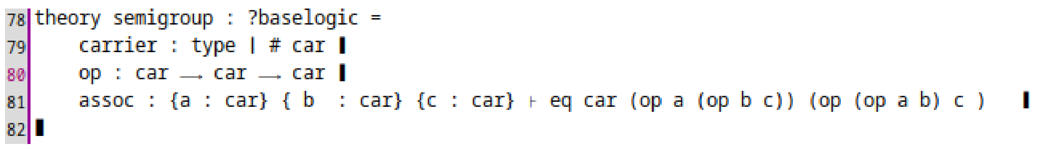
Figure 103: trivial yet cumbersome proof

property (type of congruence : $\{\ x\ y\ P\ \}\ x = y \to P\ x \to P\ y\ )$.
When the setup described above is done the rewrite tactic modifies the proof state accordingly (this is where the *in*-parameter matters) and generates the proof term.

6 Algebraic equality reasoning

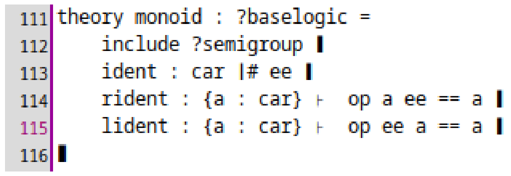
6.1 Motivation

Proving arithmetic expressions is cumbersome. For example does the theorem that $a+b+c+d = d+(b+(a+c))$ (where a, b, c, d are integers) require



```
theory semigroup : ?baselogic =
    carrier : type | # car ❙
    op : car ⟶ car ⟶ car ❙
    assoc : {a : car} { b  : car} {c : car} ⊦ eq car (op a (op b c)) (op (op a b) c )   ❙
❚
```

Figure 104: implementation of the abstract representation of the semigroup structure



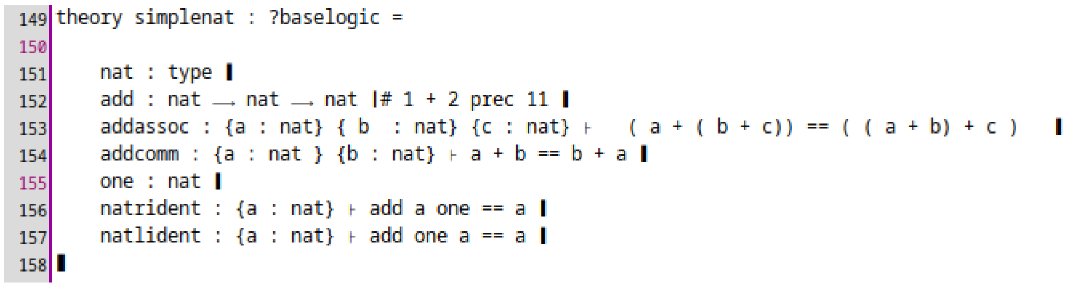
```
theory monoid : ?baselogic =
    include ?semigroup ❙
    ident : car |# ee ❙
    rident : {a : car} ⊦  op a ee == a ❙
    lident : {a : car} ⊦  op ee a == a ❙
❚
```

Figure 105: implementation of the abstract representation of the group structure

many trivial steps until it is proven. For such situations algebraic/arithmetic automation is desirable to remove busywork from the user. The complete manual proof is given in fig. 103 using Coq. As one can see, this only requires simple, repetitive steps. Through experience it became apparent that such simple algebraic/arithmetic proofs often occur. The consequence is then that the general flow of a proof gets disturbed by these nuisances.
Due to being repetitive, simple in theory and a thing that very common, simple arithmetic/algebraic should be done automatically so that the user can focus on the important parts of a proof. In case of the example in fig. 103 Coq offers a tactic, named **ring**, which solves the proof with just two tactic calls (the other one being the *intros*-tactic since **ring** can't handle quantifiers).

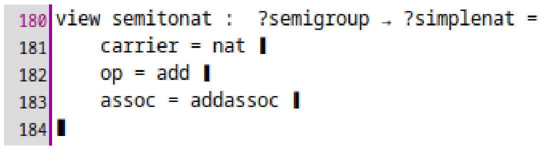
6.2 System description

The system is made up of a small hierarchy of classes. On top is the so called Algebra-tactic which takes a bunch of smaller tactics that implement a solver for an algebraic structure on an abstract level (i.e. not for a concrete type like *nat*, but for example an abstract semi-ring). The strategies in turn collect instantiations for the algebraic structure they work on. For example when there is a strategy for solving semigroup structures, instances would include an interpretation of semigroup structure in the natural numbers (i.e. the



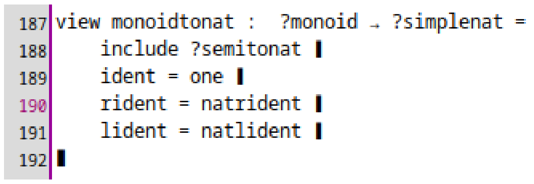
```
theory simplenat : ?baselogic =

    nat : type ❙
    add : nat ⟶ nat ⟶ nat |# 1 + 2 prec 11 ❙
    addassoc : {a : nat} { b  : nat} {c : nat} ⊦   ( a + ( b + c)) == ( ( a + b) + c )   ❙
    addcomm : {a : nat } {b : nat} ⊦ a + b == b + a ❙
    one : nat ❙
    natrident : {a : nat} ⊦ add a one == a ❙
    natlident : {a : nat} ⊦ add one a == a ❙
❚
```

Figure 106: simple sample implementation of the natural numbers



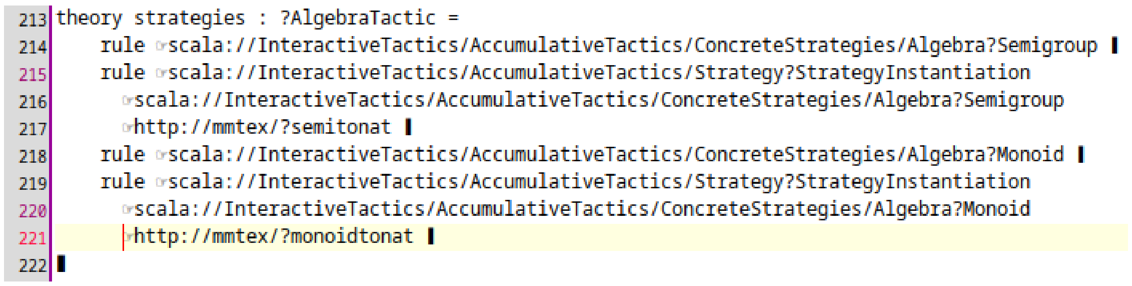
```
view semitonat :  ?semigroup → ?simplenat =
    carrier = nat ❙
    op = add ❙
    assoc = addassoc ❙
❚
```

Figure 107: simple sample implementation of the natural numbers



```
view monoidtonat :  ?monoid → ?simplenat =
    include ?semitonat ❙
    ident = one ❙
    rident = natrident ❙
    lident = natlident ❙
❚
```

Figure 108: simple sample implementation of the natural numbers



```
theory strategies : ?AlgebraTactic =
    rule ☞scala://InteractiveTactics/AccumulativeTactics/ConcreteStrategies/Algebra?Semigroup ❙
    rule ☞scala://InteractiveTactics/AccumulativeTactics/Strategy?StrategyInstantiation
      ☞scala://InteractiveTactics/AccumulativeTactics/ConcreteStrategies/Algebra?Semigroup
      ☞http://mmtex/?semitonat ❙
    rule ☞scala://InteractiveTactics/AccumulativeTactics/ConcreteStrategies/Algebra?Monoid ❙
    rule ☞scala://InteractiveTactics/AccumulativeTactics/Strategy?StrategyInstantiation
      ☞scala://InteractiveTactics/AccumulativeTactics/ConcreteStrategies/Algebra?Monoid
      ☞http://mmtex/?monoidtonat ❙
❚
```

Figure 109: simple sample implementation of the natural numbers

carreir are the natural numbers with the addition as the semigroup operator), integer, lists (the operator is the append function).
When a strategy defines its solving procedure for a given algebraic structure, that structure has first to be defined inside an MMT theory. The strategy then references the abstract operators, carrier etc. from that theory in its solving procedure.
An instantiation is a view from the theory defining the abstract algebraic structure to a theory that works on concrete types (for example the natural numbers) passed to the *strategy generation rule* which will then register that instantiation with the indicated strategy (the name of the target strategy is also passed to that rule; every strategy has a global name).
A strategy itself is a Scala code snipped that gets included by using the *StrategyInstantiation*-rule in MMT.
The algebra itself just asks all his registered strategies in descending order of priority (a parameter that every strategy has) whether it is applicable to the current goal. If the strategy indicates that it is, it then gets executed by the algebra tactic. If the strategy fails or indicates that it can't be applied to the current goal then the algebra tactic tries the next strategy. The algebra tactic also takes an optional extra argument which lets the user force the algebra tactic to try a specific strategy like for example *semigroup*.

Example Let there be two strategies for *semigroup* and *monoid* as displayed in 104 and 105. Now to register a instantiation for natural numbers (here some fake natural numbers as in 106 are used) for the two strategies one has to make a view from *semigroup*/*monoid* to *simplenat* (see 107 and 108). Now to create a strategy instantiation one calls the parametric rules as shown in 109 (lines 215 to 217 and lines 219 to 221) passing the global name of the target strategy and then the global name of the view. One has to make the system first aware of the strategies though (therefore the rule calls in the lines 214 and 218).

6.2.1 Accumulative tactics

The abstract concept of accumulative tactics is what the algebra tactic is build upon. Therefore a closer look at them increases the understanding of how the algebra tactic and its strategies work. A general overview of the accumulative tactics is given in 110. The whole accumulative tactics/strategy complex is implemented as a hierarchic system of rules. On top is the rule

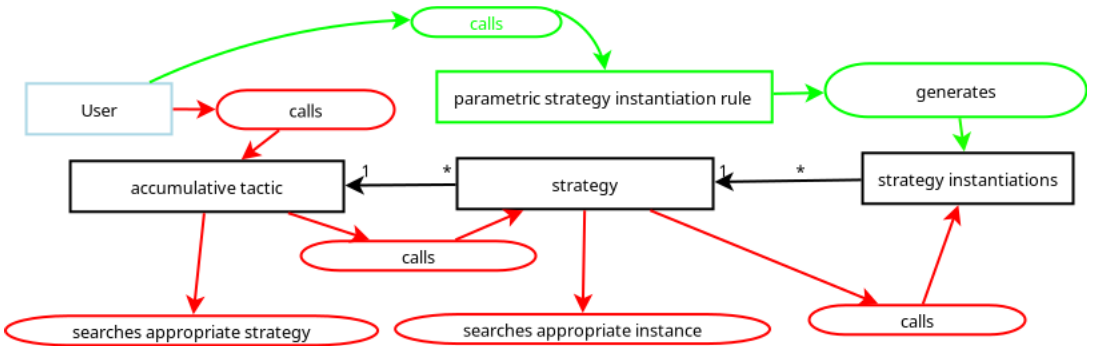
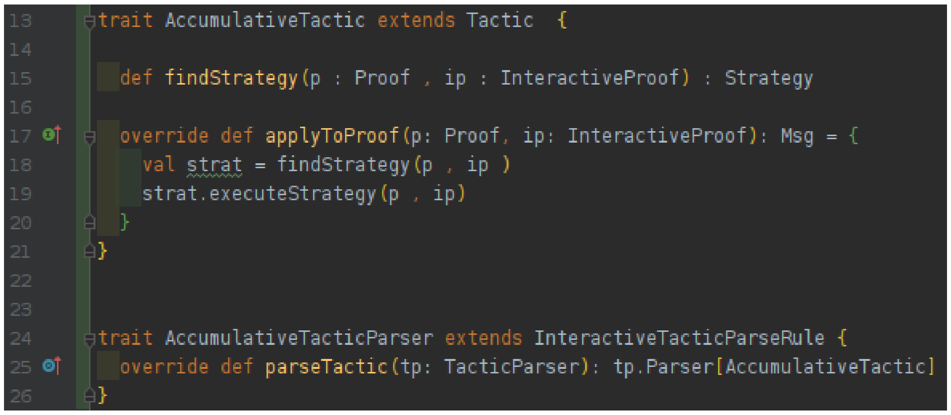

calls
User
calls
parametric strategy instantiation rule
generates
accumulative tactic
1
*
strategy
1
*
strategy instantiations
calls
searches appropriate strategy
searches appropriate instance
calls

Figure 110: general realtion between accumulative tactics, strategies and strategy instances

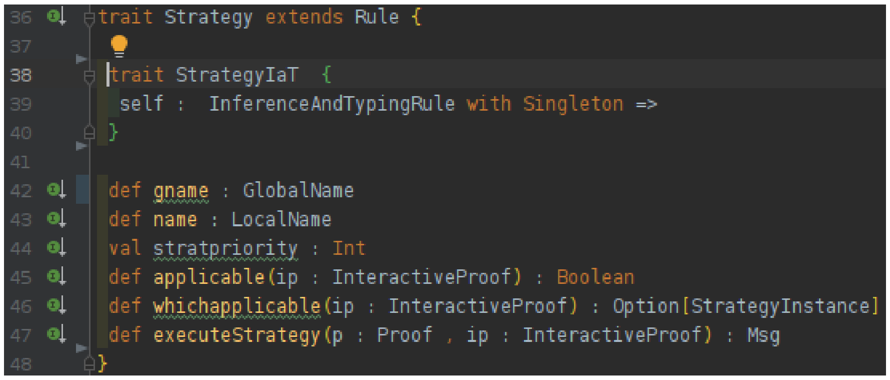


```
trait AccumulativeTactic extends Tactic  {

  def findStrategy(p : Proof , ip : InteractiveProof) : Strategy

  override def applyToProof(p: Proof, ip: InteractiveProof): Msg = {
    val strat = findStrategy(p , ip )
    strat.executeStrategy(p , ip)
  }
}


trait AccumulativeTacticParser extends InteractiveTacticParseRule {
  override def parseTactic(tp: TacticParser): tp.Parser[AccumulativeTactic]
}
```

Figure 111: The trait for the AccumulativeTactic and AccumulativeTactic-Parser

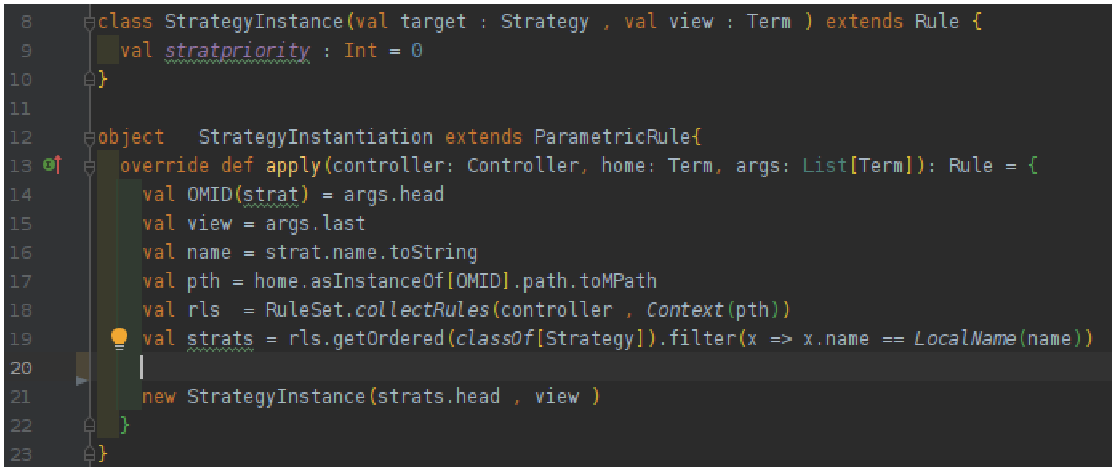
for the accumulative tactics, followed by the strategies and at the bottom are the strategy instantiations.

Accumulative tactic The accumulative tactic is a super structure that coordinates different so called strategies. The role of a accumulative tactic is more to decide when to use a certain so called **strategy**. If necessary though, it can partake in the actual proof process but is supposed to let the selected strategy to do the bulk of the work.
To make an own accumulative tactic one simply extends from the *AccumulativeTacticParser*-trait (fig. *acctrait*) and defines a parser which will return



```
trait Strategy extends Rule {

 trait StrategyIaT  {
  self :  InferenceAndTypingRule with Singleton =>
 }

 def gname : GlobalName
 def name : LocalName
 val stratpriority : Int
 def applicable(ip : InteractiveProof) : Boolean
 def whichapplicable(ip : InteractiveProof) : Option[StrategyInstance]
 def executeStrategy(p : Proof , ip : InteractiveProof) : Msg
}
```

Figure 112: The trait for the Strategy



```
class StrategyInstance(val target : Strategy , val view : Term ) extends Rule {
  val stratpriority : Int = 0
}

object   StrategyInstantiation extends ParametricRule{
  override def apply(controller: Controller, home: Term, args: List[Term]): Rule = {
    val OMID(strat) = args.head
    val view = args.last
    val name = strat.name.toString
    val pth = home.asInstanceOf[OMID].path.toMPath
    val rls  = RuleSet.collectRules(controller , Context(pth))
    val strats = rls.getOrdered(classOf[Strategy]).filter(x => x.name == LocalName(name))

    new StrategyInstance(strats.head , view )
  }
}
```

Figure 113: The parametric rule used for generating the instantiations of strategies

an *AccumulativeTactic* (fig 111). To make the tactic available one then references the concrete instance of the *AccumulativeTacticParser* via MMT's rule mechanism.
Implementations of the *AccumulativeTactic* are forced to implement the findStrategy method which will then go through a list of strategies and returns a lazy list of applicable strategies. Which then get executed one after the other until one succeeds or all failed.

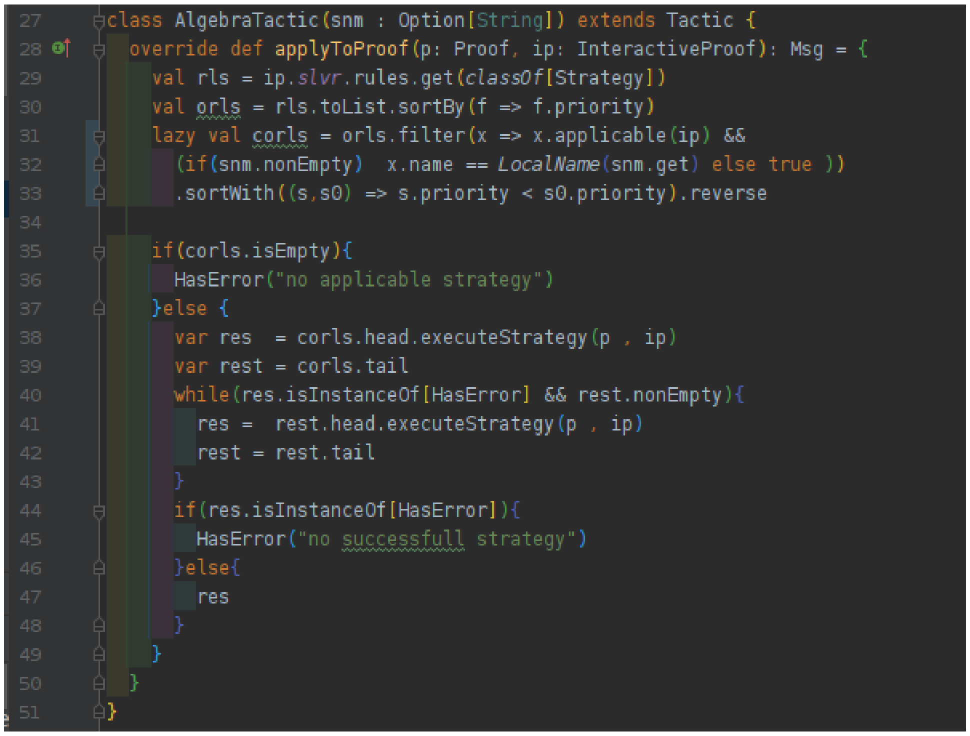
Strategy Strategies implement the *Strategy*-trait (fig. 112) which will force concrete classes to implement a bunch of meta information and the functions *whichapplicable*, which searches for a strategy instantiation that can be used to work on the current goal, and the *executeStrategy*-function which is where the main functionality goes. The implementation of the *Strategy*-trait also forces the implementation of some meta information that can be used by accumulative tactics and strategy instantiations.

Strategy instantiation As already described in the system description of the algebra tactic, strategy instantiations are basically a call to a parametric rule which associates a strategy with a view (fig. 113). When the *StrategyInstantiation* parametric rule gets called, it searches for the target strategy and stores the found strategy together with the passed view in an instance of *StrategyInstance* which will be stored in the global rule set of MMT.

6.3 Implementation of the algebra tactic

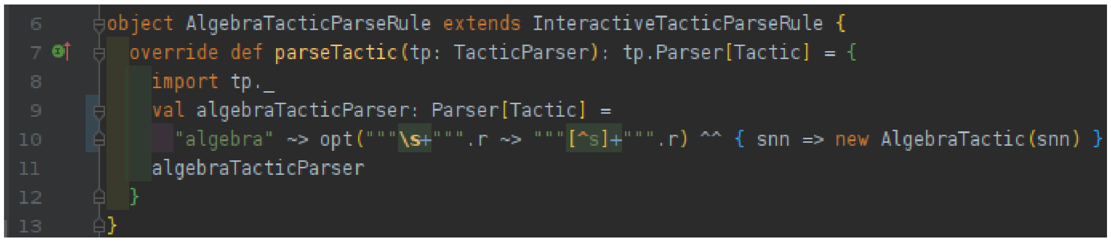
Algebra tactic The implementation of the algebra tactic is displayed in figure 114. When called it first gathers all strategies it can find and sorts them by priority. It then filters those strategies that are applicable to the goal and, if the algebra tactic was called with the optional strategy selector argument, it searches for that particular strategy and checks whether it is applicable (lines 29 to 33). If no applicable strategy was found the tactic fails with an error (lines 35 to 37). It then tries every strategy until either one succeeds (i.e. when *res.isInstaceOf[HasError]* is not true any more) or every one has failed (lines 38 to 43).

semigroup The semigroup implementation is displayed in 116. The semigroup's *whichapplicable*-function which is called by the *applicable*-function



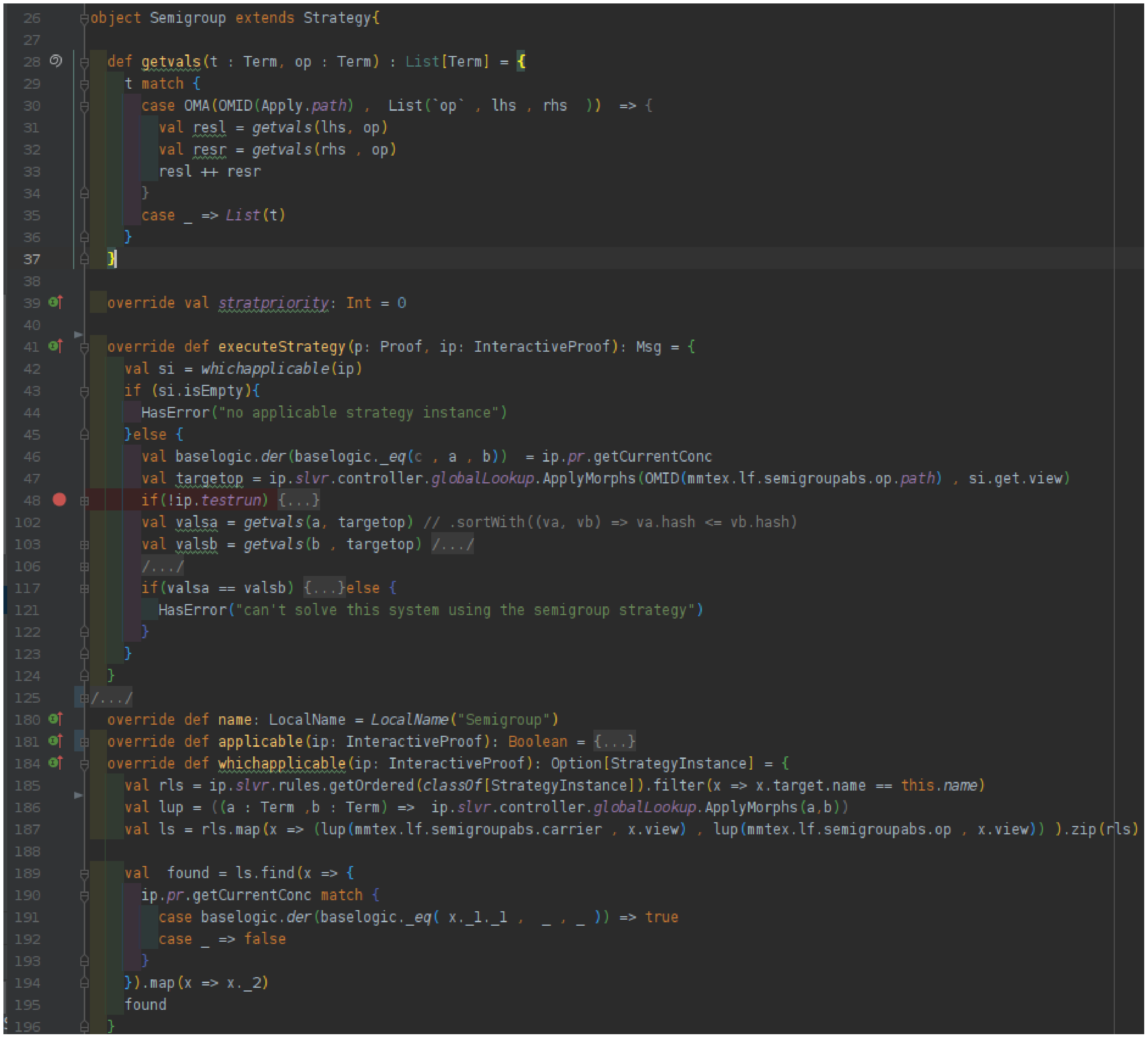
```
class AlgebraTactic(snm : Option[String]) extends Tactic {
  override def applyToProof(p: Proof, ip: InteractiveProof): Msg = {
    val rls = ip.slvr.rules.get(classOf[Strategy])
    val orls = rls.toList.sortBy(f => f.priority)
    lazy val corls = orls.filter(x => x.applicable(ip) &&
      (if(snm.nonEmpty)  x.name == LocalName(snm.get) else true ))
      .sortWith((s,s0) => s.priority < s0.priority).reverse

    if(corls.isEmpty){
      HasError("no applicable strategy")
    }else {
      var res  = corls.head.executeStrategy(p , ip)
      var rest = corls.tail
      while(res.isInstanceOf[HasError] && rest.nonEmpty){
        res =  rest.head.executeStrategy(p , ip)
        rest = rest.tail
      }
      if(res.isInstanceOf[HasError]){
        HasError("no successfull strategy")
      }else{
        res
      }
    }
  }
}
```

Figure 114: Implementation of the algebra-tractic



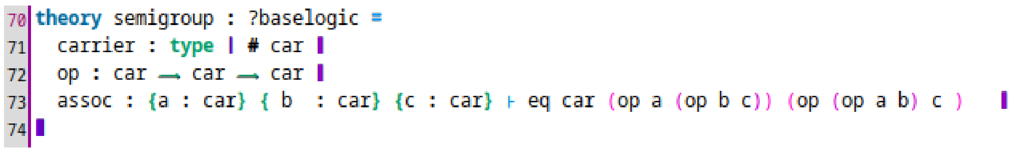
```
object AlgebraTacticParseRule extends InteractiveTacticParseRule {
  override def parseTactic(tp: TacticParser): tp.Parser[Tactic] = {
    import tp._
    val algebraTacticParser: Parser[Tactic] =
      "algebra" ~> opt("""\s+""".r ~> """[^s]+""".r) ^^ { snn => new AlgebraTactic(snn) }
    algebraTacticParser
  }
}
```

Figure 115: For the sake of completeness: the implementation of the algebra-tactic parser



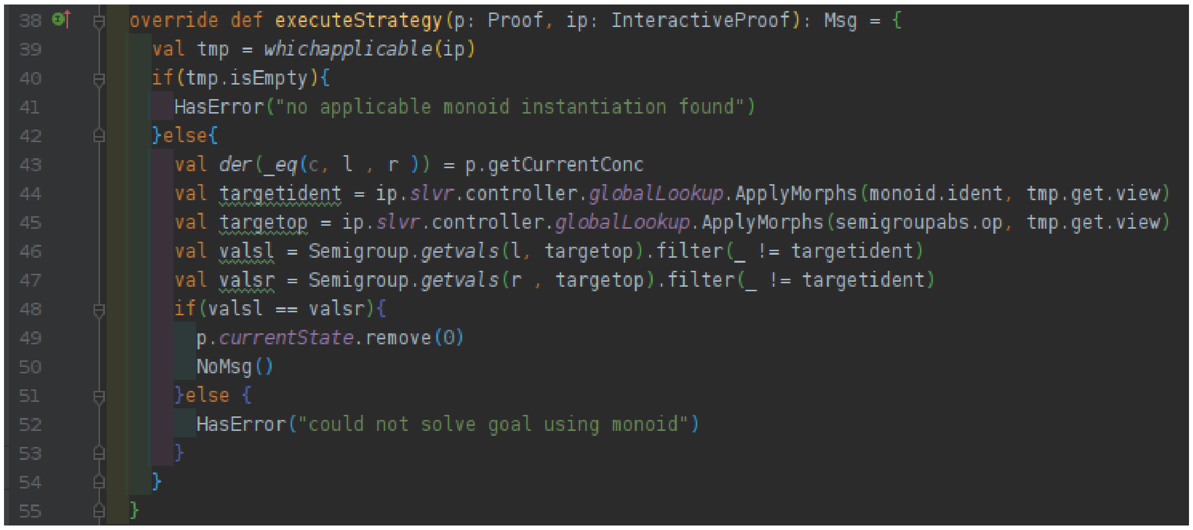
```scala
object Semigroup extends Strategy{

  def getvals(t : Term, op : Term) : List[Term] = {
    t match {
      case OMA(OMID(Apply.path) ,  List(`op` , lhs , rhs  ))  => {
        val resl = getvals(lhs, op)
        val resr = getvals(rhs , op)
        resl ++ resr
      }
      case _ => List(t)
    }
  }

  override val stratpriority: Int = 0

  override def executeStrategy(p: Proof, ip: InteractiveProof): Msg = {
    val si = whichapplicable(ip)
    if (si.isEmpty){
      HasError("no applicable strategy instance")
    }else {
      val baselogic.der(baselogic._eq(c , a , b))  = ip.pr.getCurrentConc
      val targetop = ip.slvr.controller.globalLookup.ApplyMorphs(OMID(mmtex.lf.semigroupabs.op.path) , si.get.view)
      if(!ip.testrun) {...}
      val valsa = getvals(a, targetop) // .sortWith((va, vb) => va.hash <= vb.hash)
      val valsb = getvals(b , targetop) /.../
      /.../
      if(valsa == valsb) {...}else {
        HasError("can't solve this system using the semigroup strategy")
      }
    }
  }
/.../
  override def name: LocalName = LocalName("Semigroup")
  override def applicable(ip: InteractiveProof): Boolean = {...}
  override def whichapplicable(ip: InteractiveProof): Option[StrategyInstance] = {
    val rls = ip.slvr.rules.getOrdered(classOf[StrategyInstance]).filter(x => x.target.name == this.name)
    val lup = ((a : Term ,b : Term) =>  ip.slvr.controller.globalLookup.ApplyMorphs(a,b))
    val ls = rls.map(x => (lup(mmtex.lf.semigroupabs.carrier , x.view) , lup(mmtex.lf.semigroupabs.op , x.view)) ).zip(rls)

    val  found = ls.find(x => {
      ip.pr.getCurrentConc match {
        case baselogic.der(baselogic._eq( x._1._1 ,  _ , _ )) => true
        case _ => false
      }
    }).map(x => x._2)
    found
  }
```

Figure 116: Implementation of the semigroup strategy



```
theory semigroup : ?baselogic =
  carrier : type ❘ # car ❙
  op : car ⟶ car ⟶ car ❙
  assoc : {a : car} { b  : car} {c : car} ⊦ eq car (op a (op b c)) (op (op a b) c )   ❙
❚
```

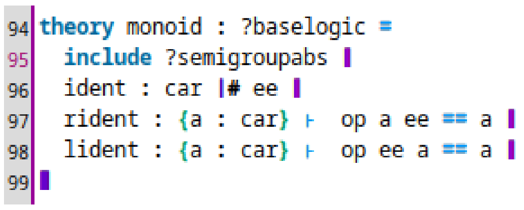
Figure 117: Implementation of the semigroup strategy



```
38  override def executeStrategy(p: Proof, ip: InteractiveProof): Msg = {
39    val tmp = whichapplicable(ip)
40    if(tmp.isEmpty){
41      HasError("no applicable monoid instantiation found")
42    }else{
43      val der(_eq(c, l , r )) = p.getCurrentConc
44      val targetident = ip.slvr.controller.globalLookup.ApplyMorphs(monoid.ident, tmp.get.view)
45      val targetop = ip.slvr.controller.globalLookup.ApplyMorphs(semigroupabs.op, tmp.get.view)
46      val valsl = Semigroup.getvals(l, targetop).filter(_ != targetident)
47      val valsr = Semigroup.getvals(r , targetop).filter(_ != targetident)
48      if(valsl == valsr){
49        p.currentState.remove(0)
50        NoMsg()
51      }else {
52        HasError("could not solve goal using monoid")
53      }
54    }
55  }
```

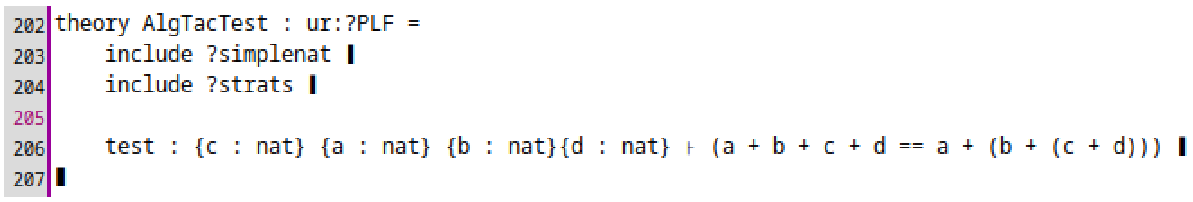
Figure 118: The essentially changed part of the monoid-implementation compared to the semigroup strategy

first retrieves all its strategy instances (line 185). Then it sets up a list of tuples containing the carriers and the rule each belongs to (lines 186 and 187). Finally the whichapplicable-function searches though this list of tuples to find pair that that can be used on the current goal (lines 189 to 195).
The *executeStrategy* function first gets the selected strategy instance (line 42) and then takes the current goal, which has to be of shape $lhs = rhs$ (line 46). It then retrieves the binary operator that combines two values of the carrier (line 47) by applying the view (that is the view morphism) that is saved in the selected strategy instance to the abstract definition of *op* (as shown in fig. 117). This gets then used to retrieve the values from the lhs and rhs as a list each by calling the *getVals*. Finally, when the lists are both equal the system removes the current goal as solved.



```
theory monoid : ?baselogic =
  include ?semigroupabs ❙
  ident : car |# ee ❙
  rident : {a : car} ⊦  op a ee == a ❙
  lident : {a : car} ⊦  op ee a == a ❙
❚
```

Figure 119: The MMT-theory representing the monoid



```
theory AlgTacTest : ur:?PLF =
    include ?simplenat ❙
    include ?strats ❙

    test : {c : nat} {a : nat} {b : nat}{d : nat} ⊦ (a + b + c + d == a + (b + (c + d))) ❙
❚
```

Figure 120: Theory in which the *Example 1* resides

monoid The main thing that changed in the *monoid*-solver is its execution algorithm. In line 44 the strategy retrieves the concrete neutral element from the strategy instance by applying the view morphism the instance holds to *ident* from 119. This will then get used in lines 46 and 47 to remove the neutral element from the value lists the *getVals* function generates.

group Group is implemented very similarly to the semigroup strategy and monoid strategy. In general the main difference is that as an additional step, tries to eliminate inverse pairs i.e. *a* - *a*. The rest is similar to the *monoid* strategy.

6.4 Examples concerning the use of the algebra tactic

Example 1 The goal is to poof the simple theorem $\{c : nat\}$ $\{a : nat\}$ $\{b : nat\}\{d : nat\} \vdash (a + b + c + d == a + (b + (c + d)))$. This theorem can be solved incorporating the *semigroup*-strategy. After the proof is loaded, first one introduces the PIs to the proof context via the *fixs*-tactic and then executes the *algebra*-tactic. If one desires one can explicitly instruct the *algebra*-tactic to explicitly use the *semigroup*-strategy by typing “algebra semigroup”.

Example 2 The goal is to proof a more complex theorem that can be solved by group but not by the *semigroup* or *monoid* strategy. The goal is to poof the simple theorem $\{c : nat\}$ $\{a : nat\}$ $\{b : nat\}\{d : nat\} \vdash$ *(a - a + ident + b + c + d == a + ident + ident (b + (c + d))) - a* (ident it the identity element/neutral element). Since this term includes the inverse of *a* (namely - *a*) this can not be solved by *semigroup* or *monoid* either but can be solved by the *group strategy*. To solve this goal one simply types "algebra group" or just "algebra" where the tactic will find out by itself what strategy to use.

7 Future work

7.1 Implementation of truly MMT-parseable languages

7.1.1 Current state

The current way to save proofs in MMT is to just simply use the proof language from the interactive proof tool and extend MMT so that one can write the proof directly as a definition into an MMT file.
Using the interactive tactics language to document proofs comes with some disadvantages though. In general tactic languages aren't very readable. Usually the proof state isn't mentioned explicitly. This makes tracking the proof steps fairly hard, if not impossible. On top of that tactics are usually not self explanatory. This means that one has to put in extra effort to understand a proof, even when the proof state before and after the execution of the tactic is known. Additionally the tactics language used for the interactive proofs does a lot of things implicitly like the "fixs" tactic (without explicit arguments) which introduces all hypothesis to the local proof context while giving the added hypothesis names automatically. Then, later in the proof those hypothesis can be referred to by their implicitly given name. As a result the proofs become even more unreadable. The just mentioned fact that fixes introduces names automatically if wished shows another problem with that language. The interactive tactics language ignores established MMT conventions (or to be more precise goes beyond the current capabilities of MMT). In MMT, variables have first to be declared *explicitly* before they can be referenced down the line.

Implementation In its current state MMT is unable to parse the interactive version of the tactics language properly. As described above this has to do with the fact that the interactive tactics language allow for implicitly introduced variables. When MMT encounters a constant it checks its local and global context whether there is a constant/variable named exactly like the atom. Now the problem is that new names must be introduced either by a constant definition in the global context or by a binding construct in the local context. The tactics language though sometimes introduces new names by means that go beyond that. Therefore, a custom parser was made so that whenever a tactics proof is encountered in the MMT file (a proof is marked by the enclosing *proof* and *qed*key words) first just parses the proof as one long string. During type checking, when the proof is type checked, the string then gets actually parsed by an external parser (the one used for the interactive front end) and then immediately evaluated. The main disadvantage is that MMT is basically blind to the actual structure of the proof since it only recognizes the proof as one long string which contradicts MMT and its targets to be a system that links different knowledge sources together via its OMDOC format.

Parseable langauges To be true to MMT's original purpose as interlink language it is necessary to be able to represent tactic proofs in a way that MMT can actually parse. Two approaches have been investigates so far: a modification of the interactive tactics language which mainly removes problematic constructs from the interactive language, like implicit introduction of variables (i.e. all variables have to be introduced properly before they can be used). This approach has been worked on a bit so far and looks promising. Its main advantage is that it can be completely implemented with the capabilities the MMT system offers without the need for Scala side extensions to the MMT system (that is for the parsing part, the type checking still has to be diverted to the external prover).
The second approach would be to implement a very explicit structured proof language in the style of Isabel's Isar. The main advantage of such a solution would be that proofs become readable even without the usage of an interactive system. The disadvantage is that one needs to implement parser extensions to the MMT system which requires Scala-level programming since the syntax is to complex to be implemented with MMT-only means. It is however the goal to be so explicit that it can be translated into a properly

parsed structure instead of being just a string from the viewpoint of MMT (i.e. explicit naming of variables and avoidance of other problematic structures).

7.2 Implementation of the proof term generation for the algebra strategies

Due to time constraints the algebra strategies were implemented without the part that generates the proof term, instead the algebra strategies merely modify the proof state. Therefore the algebra tactics is only usable in test-mode in the interactive proof gui.
The main problem is to develop an algorithm that is generic enough to solve arbitrary semigroups, monoids, rings etc. These are not trivial undertakings, quite the contrary, these algorithms are fairly complex and go beyond the scope of a masters thesis (especially implementing multiple different ones). Therefore learning form well established systems like Coq and Isabelle (i.e. reading their source code, published papers about algebraic automation) is key. Getting used to theses systems alone will take a good amount of time, first, because understanding complex systems like Isabelle, Coq, MMT, etc. takes in and of itself a long time and second because the actual solver algorithms aren't all that well documented.

7.3 Other missing implementations

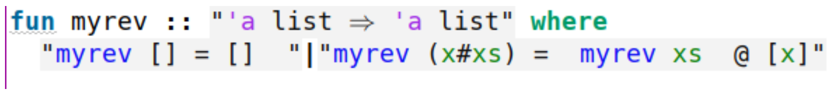
Due to time constraints it was not possible to implement a ring-strategy for the algebra tactic and the heterogeneous transitivity proof chains.
The ring tactic suffers from the problem that it needs to be an abelian group and a semi group for the same carrier but for different binary operators (additive operator and multiplicative operator). Due to the way the abstract algebraic structures were implemented on the MMT side this requires to define the group structure a second time for the additive operator because otherwise the semigroup that ring imports and the group structure that ring imports (which builds upon the semigroup definitions) have the same binary operator. Defining the group structure a second time is not a clean solution though and parametric view don't exist yet. Therefore another solution is needed and has to be investigated for.
The heterogeneous chains should be fairly straight forward with regards to

their implementation because most of the functionality is easily implemented using *InferenceAndTyping*-rules.

7.4 Traceable tactics/automation

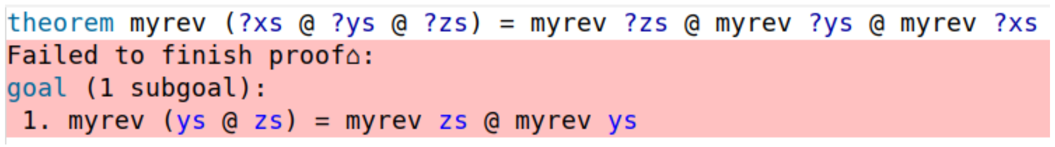
Often it is hard to for a user (especially new ones) to understand why a certain tactic failed. This is especially true for automation that doesn't represent just a basic proof step but performs a multitude of proof steps. Sometimes automation doesn't perform classical proof steps as in "executing a tactic". An example of such a "tactic" would be the Metis proof method in Isabelle/HOL which translates the to be proven into a first order logic problem which Metis then tries to solve. In case Metis can solve the problem the proof is then "just solved" without further explanation. This is in so far bad since the user potentially wants or needs to know how the proof was conducted (especially in a learning/teaching scenario where the autosolver partially replaces a tutor). For this case, where Metis successfully proofs that a statement is true, Isabelle tries to reconstruct the Metis-proof as Isar-proof (when invoked via Isabelle's "Sledgehammer"). This kind of backtranslation is kind of lack luster though. First the reconstruction of the Metis-proof in Isar doesn't always work, second, even if it works the proof is usually "designed" in a way that is not intuitive to humans.
Another problem arises when a failing proof does not offer traceability. Again looking at Metis, when a proof using Metis fails the user is only informed that the proof failed, but not why. Usually when a proof fails it is even more important to have the ability to see what proof steps have been taken in the failing proof so that the user can debug the proof. Without such capabilities the user is left with educated guesses why the proof has failed.
For the above stated reasons proof assistants like Coq, Isabelle and ACL2 (among others) offer different tracing mechanisms to allow the user to understand why an (automated) tactic has failed (or succeeded).

Isabelle Isabelle is fairly heavy on the "show don't tell" part when it comes to its automation. Non the less it offers some limited capabilities that allow the user to retrace what happened.



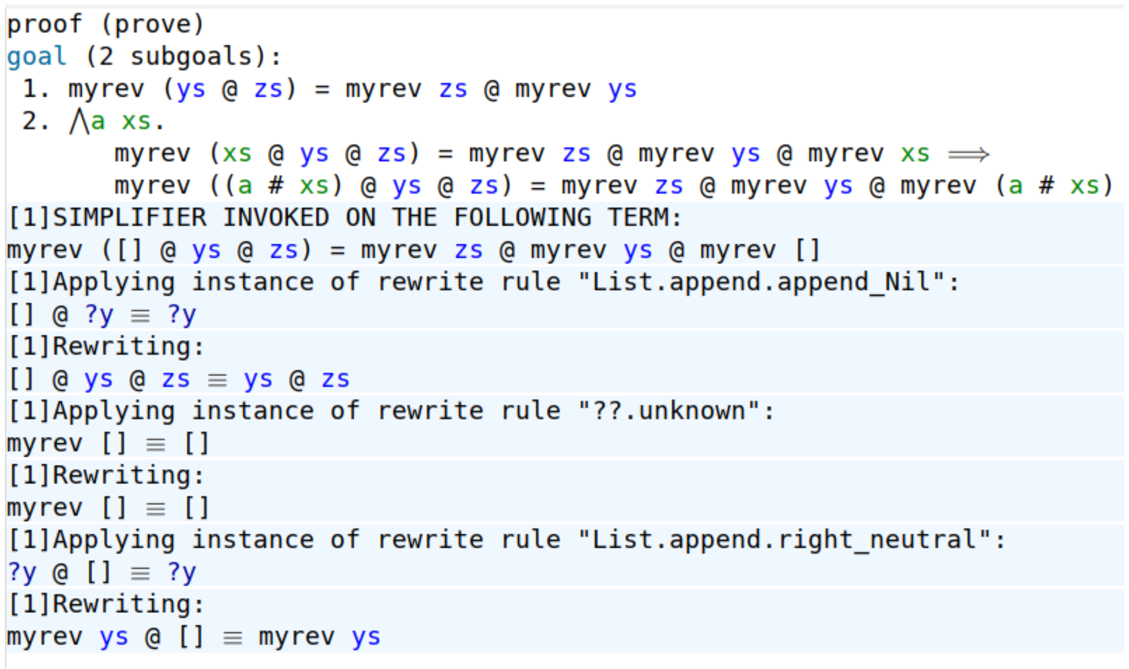
```
fun myrev :: "'a list ⇒ 'a list" where
  "myrev [] = []  "|"myrev (x#xs) =  myrev xs  @ [x]"
```

Figure 121: definition of the myrev function



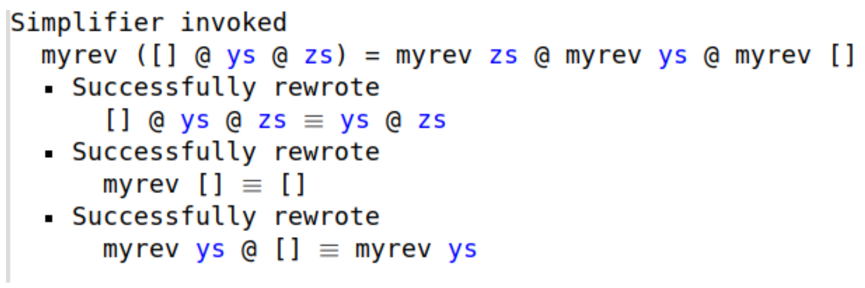
```
theorem myrev (?xs @ ?ys @ ?zs) = myrev ?zs @ myrev ?ys @ myrev ?xs
Failed to finish proof⌂:
goal (1 subgoal):
 1. myrev (ys @ zs) = myrev zs @ myrev ys
```

Figure 122: This proof fails because the lemma *myrev (xs @ ys) = myrev ys @ myrev xs* is missing
. Therefore *simp* can only proof the base case of the induction (without explicitly stating induction this proof would just be stuck at the very beginning)



```
proof (prove)
goal (2 subgoals):
 1. myrev (ys @ zs) = myrev zs @ myrev ys
 2. ⋀a xs.
       myrev (xs @ ys @ zs) = myrev zs @ myrev ys @ myrev xs ⟹
       myrev ((a # xs) @ ys @ zs) = myrev zs @ myrev ys @ myrev (a # xs)
[1]SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
myrev ([] @ ys @ zs) = myrev zs @ myrev ys @ myrev []
[1]Applying instance of rewrite rule "List.append.append_Nil":
[] @ ?y ≡ ?y
[1]Rewriting:
[] @ ys @ zs ≡ ys @ zs
[1]Applying instance of rewrite rule "??.unknown":
myrev [] ≡ []
[1]Rewriting:
myrev [] ≡ []
[1]Applying instance of rewrite rule "List.append.right_neutral":
?y @ [] ≡ ?y
[1]Rewriting:
myrev ys @ [] ≡ myrev ys
```

Figure 123: trace of the simplifier applied to the base case of the induction



```
Simplifier invoked
  myrev ([] @ ys @ zs) = myrev zs @ myrev ys @ myrev []
  ▪ Successfully rewrote
      [] @ ys @ zs ≡ ys @ zs
  ▪ Successfully rewrote
      myrev [] ≡ []
  ▪ Successfully rewrote
      myrev ys @ [] ≡ myrev ys
```

Figure 124: streamlined trace of the simplifier applied to the base case of the induction

simp mainly does simplification and proving by applying simplification rules (usually normalizing terms) and rewriting. By default, applying this proof method only results in one of two scenarios: either the proof succeeds/gets simplified or the proof fails without much information why the simplification failed (it will only display the goal state at which it got stuck; an example is given in fig. 122). Isabelle offers a simple tracing mechanism that displays a fairly "raw" overview over what the simplifier was doing (example in fig. 123). The problem with this kind of trace is that it is more akin to a stack trace in java, with more information then the usual user potentially needs to know. Further more, some information is not very well explained like the "??.unknown" rule which is an internal rule generated from the the definition of *myrev*.
Alternatively, Isabelle also offers a more streamlined version of the debug trace (example in fig. 124) which removes some of the "bloat" and presents the remaining information in a user friendly way [8].

7.5 Conclusion

This work explored the equatinal reasoning capabilities of other systems which at least to some degree hand it differently, while all offer different advantages in general they all are an improvement over writing plain lambda terms. Learning and inspired by these systems, new ways and means were added to MMT for equational reasoning. Adding equational reasoning makes MMT a lot more usable and is an important step toward making it a fully fledged proof assistant. Stuff like the rewrite tactic and algebra tactic relieve the user from annoying busywork and let the user prove a lot more theorems with a reasonable amount of work. Efforts like those undertaken in this work also help to make MMT more accessible to new users. Due to time constraints all three major parts of this work are a bit lacking though, and need to be expanded on in future works.

References

[1] using the ring solver. In *https://wiki.portal.chalmers.se/agda/Libraries/UsingTheRingSolver*, 2009.

[2] agda ring solver. In *https://agda.github.io/agda-stdlib/Tactic.RingSolver.html*, 2022.

[3] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.

[4] TU Muenchen Christoph Traut, Lars Noschinski. Isabelle rewriting library examples. In *https://isabelle.in.tum.de/library/HOL/HOL-ex/Rewrite_Examples.html*, 2022.

[5] Matthew Daggitt and Alex Rice. agda standard library source repository; monoid solver source. In *https://github.com/agda/agda-stdlib/blob/master/src/Tactic/MonoidSolver.agda*, 2022.

[6] Coq development team. Definition of equals in coq. In *https://coq.inria.fr/library/Coq.Init.Logic.html#eq*, 2022.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[8] Lars Hupel. Interactive simplifier tracing and debugging in isabelle. In *Lecture Notes in Computer Science*, pages 328–343. Springer International Publishing, 2014.

[9] Tobias Nipkow. Programming and proving inisabelle/hol, 2020.

[10] Ulf Norell. Dependently typed programming in agda. pages 1–2, 01 2009.

[11] Ulf Norell. Agda language reference; function defintion. In *https://agda.readthedocs.io/en/v2.6.2.1/language/function-definitions.html*, 2022.

[12] Ulf Norell. Agda language reference/function definitions/dot pattern. In *https://agda.readthedocs.io/en/v2.6.1/language/function-definitions.html#dot-patterns*, 2022.

[13] Ulf Norell. Agda language reference/with-abstraction/generalization. In *https://agda.readthedocs.io/en/v2.6.1/language/with-abstraction.html#generalisation*, 2022.

[14] Ulf Norell. Agda language reference/with-abstraction/rewrite. In *https://agda.readthedocs.io/en/v2.6.1/language/with-abstraction.html#rewrite*, 2022.

[15] Ulf Norell. Agda language reference/with-abstraction/usage. In *https://agda.readthedocs.io/en/v2.6.1/language/with-abstraction.html*, 2022.

[16] F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 19(4):1–43, 2018.

[17] Florian Rabe. Mmt: The meta meta tool (system description). 2020.

[18] The Coq Development Team. The coq reference manual. In *https://coq.inria.fr/refman/*. Inria, 2020.

[19] The Coq Development Team. The coq manual/adding new relations and morphisms. In *https://coq.inria.fr/refman/addendum/generalized-rewriting.html#adding-new-relations-and-morphisms*. Inria, 2022.

[20] The Coq Development Team. The coq manual/syntax for adding new morphisms. In *https://coq.inria.fr/refman/addendum/generalized-rewriting.html#coq:cmd.Add-Parametric-Morphism*. Inria, 2022.

[21] The Coq Development Team. The coq manual/syntax for adding new relations. In *https://coq.inria.fr/refman/addendum/generalized-rewriting.html#coq:cmd.Add-Parametric-Relatio*. Inria, 2022.

[22] The Coq Development Team. The coq manual/the ring and field tactics. In *https://coq.inria.fr/refman/addendum/ring.html*. Inria, 2022.

[23] The Coq Development Team. Reasoning with equalities. In *https://coq.inria.fr/refman/proofs/writing-proofs/equality.html*. Inria, 2022.

[24] The Coq Development Team. rewrite syntax. In *https://coq.inria.fr/refman/proofs/writing-proofs/equality.html#rewriting-with-leibniz-and-setoid-equality*. Inria, 2022.

[25] Lawrence C. Paulson Tobias Nipkow and Markus Wenzel. Isabelle/hol tutorial. In *https://isabelle.in.tum.de/doc/tutorial.pdf*, 2021.

[26] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda/Equalty.* July 2020.

[27] Markus Wenzel and Tu Munchen. The isabelle/isar reference manual. page 212, 03 2001.

[28] Markus Wenzel and Tu Munchen. The isabelle/isar reference manual. pages 211,212,213, 03 2001.

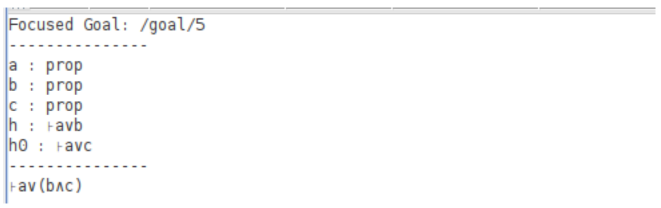
[29] Markus Wenzel and Tu Munchen. The isabelle/isar reference manual. page 300, 03 2001.

[30] Markus Wenzel and Tu Munchen. The isabelle/isar reference manual. pages 300,301, 03 2001.

[31] Sven Wille. Towards an interactive proof system in mmt. In *https://gl.mathhub.info/SvenWille/masterproj-backup-jedit/-/blob/master/masterproj/masterproj.pdf*, 2021.

A Small tutorial

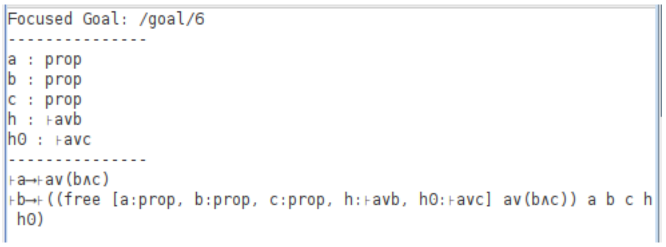
The goal is to verify that $\{a : prop, b : prop, c : prop\} \vdash a \vee b \Rightarrow a \vee c \Rightarrow a \vee (b \wedge c)$ holds. The proof goes as follows: *fixs; bwd impI; asm; bwd impI; asm; bwd (orE _ _ _ h); simp; asm; bwd orIl; use h1; simp; asm; bwd (orE _ _ _ h0); simp; asm; bwd orIl; use h2; simp; asm; bwd orIr; bwd andI; use h1; use h2*.
In plain words, this proof is done by using or-elimination on one of the disjunctions. When working on the case with “a” as hypothesis, the goal is solved trivially (using or-introduction-left). When the hypothesis is not “a” then or-elimination is applied to the other disjunction. Again, the case with a as hypothesis is solved by using or-introduction-left. In the othe case, when one has “b” and “c” as hypotheses, one uses and-introduction to create a conjunction with “b” and “c”. Then the goal is solved by or-introduction-right using this conjunction.
The proof is again started by introduction all hypotheses to the local context. Because this is a common task there is a special tactic that does just that. By using *fixs* without any arguments it takes as many hypotheses from the conclusion and adds them to the local context. Becuase the type to be proven includes object level implication fixs can only take the Pi’s and add them to the local context. To turn an object level implication into an LF-implication one simply applies the *impI* rule backwards by applying *bwd impI* (bwd unifies the conclusion of the rule with the goal, more on the backward-tactic, when the or-elimination is applied). Then the leftmost assumption can be added to the local context via asm (or fix). The second implication is handeled the same way by applying *impI* backwards and then executing asm/fix (**??**). The next step is to apply or-elimination to one of the disjunctions. For this example h is chosen (but it would work almost exactly the same with h0). The rule itself is called orE. To see what type orE has one can enter the command *gettype orE*. The type will then be printed to the output window. It would be possible to add orE to the local context via let and then apply all the needed arguments to it. But this would be a lot of unnecessary typing since there is a way the system can infer the needed arguments on its own. The first two arguments can be inferred from the fourth argument. The third argument is equal to the conclusion. The tactic appropriate for this kind of situation is the backward tactic. This tactic takes a rule and applies it backwards by unifying the conclusion of the rule with the goal. The used rule can also be an application of arguments to a rule. If an argument should be



```
Focused Goal: /goal/5
--------------
a : prop
b : prop
c : prop
h : ⊢avb
h0 : ⊢avc
--------------
⊢av(b∧c)
```

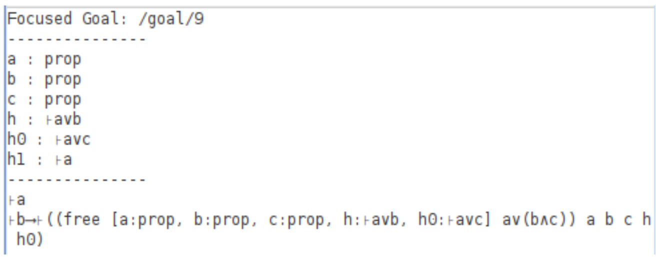
Figure 125: fixs; bwd implI; asm; bwd implI; asm

inferred by the system itself it can be marked with an underscore. Therefore, the argument for the backward tactic *bwd* is (orE _ _ _ h). The first three arguments should be inferred by the system. The fourth is needed, as stated before, to infer the first two. The third gets inferred by unifying with the goal. The remaining, not applied arguments get turned into goals. Due to a bug these goals contain the "free"-binder. To get rid of these one simply has to execute the simp tactic (simplification) (126).
Now one can use the hypothesis "h1 : ⊢ a" (previously adding it with *asm* to the local context) to solve the goal. To do so one use or-introduction-left to construct the goal. Again this can be done comfortably by using the backward tactic *bwd orIl* (127). This sub-goal can now be concluded with *use h1*. After finishing this sub-goal, the system automatically selects the next goal. Again, by using *simp* the goal becomes much simpler (128). Now one needs to again apply or-elimination. But this time using the other disjunction. As before, this first goal is solved trivially with "⊢ a" in the context and works identically as before (the remaining proof is shown in fig. 129). After the hypothesis of type "⊢ c" has been added to the local context (*asm*) the goal can be solved by this time construction the right argument of the goal. This works similar to before by applying *bwd orIr* (130). Finally, the goal is solved by constructing the conjunction of "b" and "c". Again using the backward tactic proofs to be the most convenient way. By first applying the and-introduction rule backward *bwd andI* the goal gets split into two trivial sub-goals (131) which can both be solved easily with *use h1* and *use h2*.



```
Focused Goal: /goal/6
--------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
--------------
⊦a→⊦av(b∧c)
⊦b→⊦((free [a:prop, b:prop, c:prop, h:⊦avb, h0:⊦avc] av(b∧c)) a b c h
 h0)
```

Figure 126: bwd (orE _ _ _ h); simp



```
Focused Goal: /goal/9
--------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
h1 : ⊦a
--------------
⊦a
⊦b→⊦((free [a:prop, b:prop, c:prop, h:⊦avb, h0:⊦avc] av(b∧c)) a b c h
 h0)
```

Figure 127: asm; bwd orIl

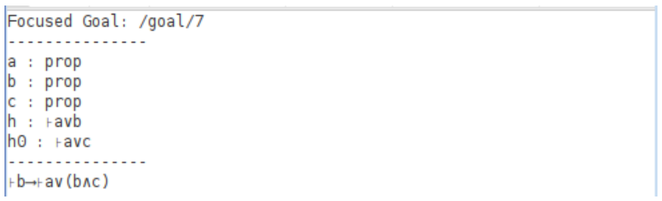
B Tactics list

This appendix presents tactics that were added to the interactive proof system while working on the main extensions presented in this work.

B.1 algebra

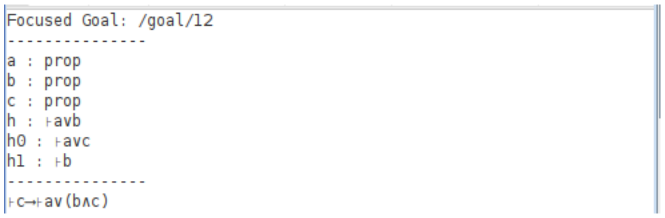
algebra <*name of strategy*> applies the algebraic strategy specified in *name of strategy* to the currently focused goal. In case no name is specified the algebra tactic tries to figure out the appropriate strategy.

Focused Goal:	/goal		
...			
a:	nat		
b:	nat		
c:	nat	$\xRightarrow{\text{algebra group}}$	goal solved
op:	nat → nat → nat		
...			
⊢ op (op a b) c = op c (op a b)			



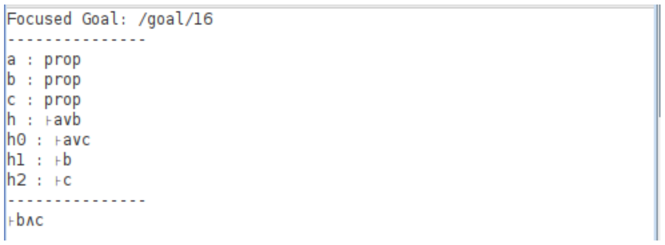
```
Focused Goal: /goal/7
---------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
---------------
⊦b→⊦av(b∧c)
```

Figure 128: use h1; simp



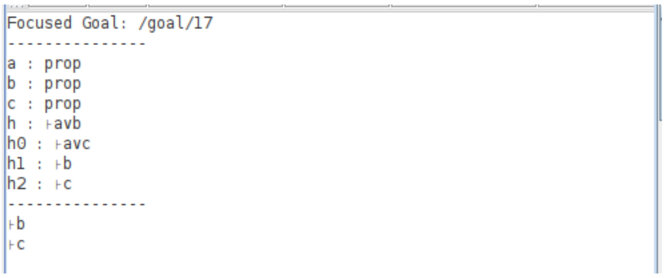
```
Focused Goal: /goal/12
---------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
h1 : ⊦b
---------------
⊦c→⊦av(b∧c)
```

Figure 129: asm; bwd (orE _ _ _ h0); simp; asm; bwd orIl; use h2; simp



```
Focused Goal: /goal/16
---------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
h1 : ⊦b
h2 : ⊦c
---------------
⊦b∧c
```

Figure 130: asm; bwd (orIr)



```
Focused Goal: /goal/17
---------------
a : prop
b : prop
c : prop
h : ⊦avb
h0 : ⊦avc
h1 : ⊦b
h2 : ⊦c
---------------
⊦b
⊦c
```

Figure 131: bwd andI

B.2 clear

clear *name* removes a hypothesis from the context. Fails if another hypothesis or the goal depends on the to be deleted hypothesis.

```
 Focused Goal:  /goal                                      Focused Goal:  /goal/0
- - - - - - - - - - - - - - - - - - -                    - - - - - - - - - - - - - - - - - - -
 ...                                                      ...
 a:                   prop              delete h          a:                   prop
 ha:                  ⊢a               ========>          ha: ⊢a
 h:                   ⊢a → ⊢a                             ...
 ...                                                     - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - -                     ⊢a
 ⊢a
```

B.3 newbackward

nbwd/newbackward term < where (*binderName* **:=** *term* | *term* **at** *position* | *binderName* **:=** *term* **at** *position*)+ > a streamlined and improved version the *bwd*-tactic. See **bwd** for a description of how the tactic generally works.

```
                                                                     Focused Goal:  /goal/0
 Focused Goal:  /goal                                               - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - -                ...
 ...                                                                 a:                   prop
 a:                   prop                                           b:                   prop
 b:                   prop                                           c:                   prop
 c:                   prop              nbwd h where (vala at 0)     d:                   prop
 d:                   prop             =========================>    vala:                ⊢ a
 vala:                ⊢ a                                            h:                   ⊢a → ⊢b → ⊢c →
 h:                   ⊢a → ⊢b → ⊢c → ⊢d                              ...
 ...                                                                - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - -                ⊢b
 ⊢d                                                                  ⊢c
```

B.4 rewrite

backward/bwd *term* performs a backward step, i.e. it takes a rule with type of the form $a \to \dots \to b$ and unifiers the conclusion of the rule with the current goal. The hypothesis of the rule are turned into new goals.

Focused Goal: /goal

...	
a:	prop
b:	prop
c:	prop
d:	prop
vala:	⊢ a
h:	⊢a → ⊢b → ⊢c → ⊢d
...	

⊢d

$\xRightarrow{\text{backward (h vala)}}$

Focused Goal: /goal/0

...	
a:	prop
b:	prop
c:	prop
d:	prop
vala:	⊢ a
h:	⊢a → ⊢b → ⊢c → ⊢d
...	

⊢b
⊢c