Analytica V: Towards the Mordell-Weil Theorem

Edmund M. Clarke [a,1] Avi S. Gavlovski [a,2]
Klaus Sutner [a,3] and Wolfgang Windsteiger [a,b,4]

[a] *Dept. of Computer Science, Carnegie Mellon University, USA*
[b] *RISC Institute, JKU Linz, Austria*

Abstract

Analytica V is a theorem proving system that is built on top of the symbolic computation system *Mathematica*. It was originally designed by E. Clarke and X. Zhao in the early 1990's. We describe here a redesign of the system that extends its abilities to reasoning about some aspects of number theory.

1 Introduction

Analytica was originally designed to reason about 19th century mathematics, in particular elementary calculus and number theory. The system was developed in the early 1990's by Xudong Zhao and Edmund Clarke, see [CZ92,BCZ98], an intermediate version of the system has been described in [CKOS03]. The system relies heavily on symbolic computation performed by *Mathematica*, the underlying computer algebra system, see [Wol02]. Apart from built-in simplification and decision procedures, Analytica also uses internal rewrite rules as well as decision procedures. For example, to deal with linear inequalities over the reals it can either exploit its own implementation of the standard elimination procedure or *Mathematica*'s linear programming algorithm. For non-linear inequalities we have implemented a variant of Bledsoe's sup-inf method.

In this paper we report on Analytica V, a redesign of the system that extends its capabilities to reason about certain concepts in number theory. In contrast to [CKOS03], where the emphasis was put on methods to connect

[1] Email: `emc@cs.cmu.edu`
[2] Email: `agavlovs@andrew.cmu.edu`
[3] Email: `sutner@aiki.ccaps.cs.cmu.edu`
[4] Email: `Wolfgang.Windsteiger@risc.uni-linz.ac.at`

Analytica to an external mathematical knowledge data base (MBase), this paper describes recent enhancements in the internal system design and the reasoning engine. As a mid-range goal, we consider the automated generation of proofs of the Mordell-Weil Theorem (the group of rational points on an elliptic curve is finitely generated) and the Dirichlet Theorem (the infinitude of primes in arithmetic progressions), see for example [Sil86]. All examples presented in this work relate to the formalization of these theorems.

As its predecessor system, Analytica *V* focuses on proofs whose logical complexity can be reduced greatly by algebraic simplification and symbolic computation, or by application of various decision procedures. For example, the proof of associtivity of the group operation on an elliptic curve relies heavily on Gröbner basis computations.

The key features of Analytica *V* that we want to describe in this work are:

- the setup of inference rules and the general proof-search mechanism,
- a look-up method for efficient check of side-conditions in computations, and
- some specialized techniques developed specifically in connection with the Mordell-Weil theorem.

We begin with a description of the Analytica *V* language and its general proof search mechanism in Section 2; the look-up mechanism is explained in Section 3. A number of specialized techniques that are relevant to the proofs shown in Section 5 are detailed in Section 4.

2 Language, Inference Rules, and Proof Search

2.1 The Analytica V Language

The object-language in the Analytica system is essentially first-order predicate logic, though we allow for restricted higher-order concepts. Notably, we allow *currying* and *universally quantified function and predicate variables in the goal*. The idea is, however, to eliminate curryed expressions by expansion of definitions given for the curryed function or predicate and to allow parameters to stand for functions and predicates. Proving goals containing existentially quantified function or predicate variables, which would require "solving for functions and predicates" in some form is not our current aim. Rather, we use currying and higher-order variables to give natural representations of mathematical concepts, notably in our examples in group-theory. The example in Section 5.1 shows typical applications of currying when dealing with group characters and homomorphisms.

Many of the terms and atomic predicates in the language carry *domain information*. Thus, we represent multiplication on the natural numbers by a term[5] `times[NN,a,b]`, whereas multiplication in an arbitrary group G is

[5] We will use *Mathematica*-syntax or *Mathematica*-oriented pseudo code when describing data structures or program fragments of Analytica *V*. In some cases, we will refer to internal

given by a term `times[G,a,b]`. In a similar vein, `eq[QQ,a,b]` denotes equality over the rationals, and so on. We have found easy access to additional domain information to be very helpful in the application of simplifiers and various decision procedures. It also helps somewhat in improving the legibility of Analytica-generated formulae.

This approach is similar, although implemented in slightly different style, to the use of functors in the Theorema system, see [Win99]. There is a complete separation between the Analytica object-level language and the *Mathematica* programming language—an aspect, in which Analytica V differs from earlier versions of Analytica. In order to have access to simplification and decision procedures provided by *Mathematica* we have a translator between the two languages.

2.2 Proofs, Proof States, and Inference Rules

The proof state always consists of a list of sequents of the form

$$\texttt{seq[}\{l_1, \ldots, l_a\}\texttt{,}\{r_1, \ldots, r_c\}\texttt{]},$$

where the l_i and r_i represent the (conjunction of) assumptions and the (disjunction of) conclusions, resprectively. The proof starts with just the initial sequent $\texttt{seq[}\{\}\texttt{,}\{g\}\texttt{]}$, where g is the goal formula to be proven. As always, a successful proof is a sequence of applications of *proof rules* that leads to an empty proof state.

All Analytica inference rules are checked into the system as pairs (T, R) where T is an applicability test and R implements the actual transformation on the proof state. T typically returns a (possibly empty) list of sequents as output. The applicability test relies heavily on pattern matching in *Mathematica.*

Proof search will be described in detail in Section 2.3. The key idea is to think of inference rules "listening for events", where by "event" we mean formulae appearing new in the current sequent. We will call the current sequent *together* with the current event our *current proof situation.* In order to relieve the individual inference rules of having to determine the actual event, the proof search will decide applicability of an inference rule by calling the associated applicability test on the current proof situation. The test can perform arbitrary tests on the proof situation, in many cases, however, pattern matching is sufficient. For efficiency reasons we provide a mechanism that allows expressions computed during the applicability test to be passed to the actual rule application in order to avoid recomputation. Therefore, any list $\{i_1, \ldots, i_t\}$ as the return value of an applicability test is interpreted as "the rule is applicable at positions $i_1, \ldots, i_t$", any other return value is interpreted as "the rule does not apply". Consequently, the implementation of an inference rule takes $i_1, \ldots, i_t$ as parameters in addition to the current sequent.

symbols used in the implementation, such as 'NN' in this case for 'the natural numbers'.

```
While[ ProofState != EMPTY,
    EnqueueRules[
        QueryRulebase[CurrentSequent[],CurrentEvent[]]];
    If[ EmptyRuleQueue[],
        DetectOutOfRules[],
        ApplyProofRule[DequeueRule[]];
    ]]
```

Fig. 1. The Analytica *V* proof search procedure.

As we will see later, it is convenient to implement an inference rule `R` using currying as `R[`$i_1, \ldots, i_t$`][seq[l,r]]`.

2.3 The Proof Search and the Prover Configuration

Analytica does not provide *one prover*. Rather, it provides a *generic proof search procedure*, which employs a global database of inference rules, the *rule base*. Arbitrary provers can be assembled by composing a rule base, i.e. a list of names of inference rules, for the proof search procedure. Apart from the rule base, the *prover configuration* consists of several global system settings that define the knowledge base relative to which a certain proof is carried out, such as definitions of predicates and functions or basic properties that are assumed to hold for a particular proof, see also Section 3.2.1 and the example in Section 5.1. The prover configuration can be adjusted for each proof individually and, in some sense, it should be seen as (part of) the "underlying theory" for the proof.

The top-level proof search itself is then rather simple, the pseudo-code given in Figure 1 is nearly identical to the actual *Mathematica* code and should almost be self-explanatory. At its center, it *queues inference rules* found by `QueryRulebase`, which simply tests all rules in the rule base using their applicability test on the current sequent with the current event. Assume the rule `R`'s applicability test resulted in $\{i_1, \ldots, i_t\}$, then `QueryRulebase` will actually return `R[`$i_1, \ldots, i_t$`]` to be stored in the rule queue. When dequeuing the rule from the queue later, it will therefore have the information passed from the rule test attached to it. `ApplyProofRule` will then apply the entire expression to the current sequent, thus resulting in `R[`$i_1, \ldots, i_t$`][seq[l,r]]` as the actual call of the inference rule `R`.

Note that we store *all rules* applicable to the current proof situation in the rule queue although we apply then only the first one. This means that, when we enqueue new rules, there might be still rules from previous cycles in the queue. We do not simply queue the new ones at the end, but we always sort the queue w.r.t. the age of a rule—a notion that takes into account the last time when a rule was applied. This strategy gives priority to the new rules queried but it prevents older rules from "starving in the queue".

Once we are faced with an empty queue, i.e. no new rules apply to the current proof situation and there are no more pending rules applicable to previous situations, we detect that we essentially "ran out of ideas". The proof does not necessarily fail at this point; rather, we increase our "level of despair", and only if that level reaches a certain threshold the proof attempt fails. We will provide certain inference rules that make their applicability test depend on the current level of despair. Hence, by gradually increasing the level of despair, an inference rule may become applicable at some point although it did not apply to the same proof situation earlier on. This seems to conform well with the approach taken by a human prover: As long as standard techniques apply, one uses those to make progress in the proof. Once one gets stuck, one considers more complicated (and, in our case, computationally more expensive) methods or methods that may not seem immediately promising. In the current implementation it is advisable to guard all inference rules that involve complicated and potentially time-consuming symbolic computations in this fashion against eager application.

In addition, there are default actions that can be performed before as well as after the rule taken from the queue. In the current status, the only rule applied before the queued rules is a normalization rule, which takes care about the propositional connectives and the logical quantifiers. For example, normalization will perform And-Splits in the antecedent.

The introduction of meta-variables[6] for existentially quantified variables in the consequent happens, mainly due to administrative reasons, in a separate rule. Note also, that case splits for disjunctions in the antecedent (Or-Left) and conjunctions in the consequent (And-Right) are not performed during normalization. We implemented separate rules for Or-Left and And-Right, because we want to split the sequent only if we are out of other rules. This is important, in particular, for instantiation of evars. Since we will try to instantiate evars in conjunctions in certain situations by symbolic computation decision procedures, e.g. algorithms for finding solutions of systems of algebraic equations or inequalities, it is beneficial to not perform the And-Split immediately.

The maintenance of the proof state and the current event, the actual application of an inference rule to the current sequent, and the maintenance of the individual sequents is accomplished by `ApplyProofRule` and its companion programs. Most importantly, we perform certain simplifications on new sequents before they enter the proof state. These simplifications can be

sequent-level simplifications, such as removing duplicate formulae, occurrences of `true`/`false` etc. from the antecedent/consequent, closing a sequent if `true`/`false` appear in the consequent/antecedent, removing formulae in the consequent that "follow easily" from the antecedent (we refer to Section 3 for when we consider a formula to "follow easily"),

formula-level simplifications, which are mainly boolean simplifications on

[6] We sometimes refer to a meta-variable as an "evar" (for "existential variable"). Introducing meta-variables for existential goals is a well-known technique used in many systems nowadays.

individual formulae, such as handling double-negation, removing duplicate subformulae in conjunctions/disjunctions and the like, or

custom simplifications, such as simplifying equalities/inequalities/disequalities with identical subterms. Additional custom simplifications to be applied at this stage—very convenient for user-defined functions or predicates—can be specified through *Mathematica* transformation rules stored in a global variable.

Having these simplifications at a central place is very helpful, since the individual inference rules can then neglect this aspect entirely, i.e. they can just compose new formulae and their simplification will be taken care of automatically.

3 Backward-Reasoning for Checking Side-Conditions

3.1 The Problem

The idea of a small, efficient backward-reasoning unit originally arose during the implementation of a simplifier for group-theory but it developed into a system component, which turned out to be applicable in many situations, both *during simplification* and *during logical reasoning.* The problem at hand is one of the central Calculemus-problems, namely the integration of reasoning techniques into computation. As an example, consider a simplification method for expressions in group-like structures to be applied during a proof. Given associativity, we want to re-arrange parentheses; if we have an identity, we want to cancel identity terms and so on. In general, when we need to check a condition C w.r.t. our current knowledge Γ, it may require a proof that C follows from Γ. However, we do not wish to employ a full-blown prover at that point: on the one hand, the argument often does not require a complicated reasoning, on the other hand it may well require a careful choice of reasoning rules—a task handled by the user in our current setting [7]. In real-world proofs, though C may not be contained in Γ, little more than unfolding of definitions is required to derive C from Γ. In the above example, we might know that "we are dealing with an abelian group". Expanding this fact we obtain all the required information. However, it is inefficient to eagerly expand definitions in Γ and apply forward-reasoning techniques for three reasons:

(i) it unnecessarily inflates Γ,

(ii) it is difficult to determine beforehand when to stop expanding, and

(iii) for the human reader of the proof, these steps are (mostly) uninteresting.

[7] Admittedly, using a standard reasoning engine that requires no user-interaction whatsoever, such as, e.g., either of the powerful resolution provers competing in every year's CASC, would be an option.

3.2 The Analytica-Solution

Our approach is to devise a simplified reasoning mechanism for looking up facts in some knowledge base Γ, which is tailored towards the needs in a particular proof setting. The key requirements are efficiency, easy extensibility (by new dependencies in the knowledge base), and easy use in its application (for the Analytica developers). Completeness has intentionally no priority, because this would lead us towards full-blown proving. Rather, we will reason only for a restricted class of goals and we will only use a very limited inference scheme, namely we will be able to look up C in Γ if

- $C \in \Gamma$ or C is known due to built-in knowledge[8], or
- C has the form $C_1 \wedge C_2$ and we can look up both C_1 and C_2 in Γ, or
- C has the form $C_1 \vee C_2$ and we can look up either C_1 or C_2 in Γ, or
- there is a formula ϕ, such that $\phi \Rightarrow C$ and we can look up ϕ in Γ.

This translates directly into a *recursive procedure* for looking up C in Γ, with the last case being worth closer inspection. Which ϕ do we choose and which implications do we employ? The idea is to specify the list of available implications as part of the prover configuration and use these implications to build up a look-up table at the beginning of the proof. Using this look-up table we can then do backward reasoning from C to all possible ϕ. As already said in Section 2.3, all formulae part of the prover configuration are assumed to hold, we do neither require them being axioms nor do we require them being proven before they may be put to the prover configuration.

3.2.1 Building the Look-up Table

We require a list of formulae available for building up the look-up table to be specified as part of the prover configuration. Only universally quantified biconditionals or implications are allowed, since these will allow backward reasoning. As for biconditionals, we need to direct each biconditional into an implication, thereby essentially throwing away half of the information. We have not investigated this case in general; at this point we only process "definitions by biconditionals" of the form[9] $\forall x : p[x] \Leftrightarrow Q$ that introduce a new predicate symbol p, and we use such defintions as $\forall x : p[x] \Rightarrow Q$. As for implications, we split up each implication $\forall x : P \Rightarrow Q_1 \wedge \ldots \wedge Q_n$ into $\forall x : P \Rightarrow Q_1, \ldots, \forall x : P \Rightarrow Q_n$ and proceed recursively. An implication $\forall x : P \Rightarrow Q$ is only processed further if its right-hand side Q is an *atomic proposition*, otherwise it will not be used for look-up purposes, i.e. the backward reasoning described above will only be done from atomic propositions. Finally, we need to process implications of the form $\forall x : P \Rightarrow Q$, where we assume that Q contains no other free variables than x, but not necessarily all of them. If P

[8] A list of built-in facts can be specified as part of the prover configuration, see the example in Section 5.1

[9] In $\forall x : P$ the x can actually stand for a sequence of variables.

and Q share the same free variables x, then we can reason backwards from *any concrete instance* of Q of the from $Q_{x \to t}$ to $P_{x \to t}$. If some variables y are free in P but not in Q, then we observe that $\forall x, y : P \Rightarrow Q$ is equivalent to $\forall x : (\exists y : P) \Rightarrow Q$ and we can reason backwards from any $Q_{x \to t}$ to $\exists y : P_{x \to t}$. The same considerations regarding free variables have been taken into account in the top-level inference rule for the realization of backchaining using implications, see the proof of the image of δ in section 5.2 for an example of backchaining.

Along these lines, we have implemented a framework that generates the recursive look-up procedure—in form of recursive definitions for a *Mathematica* program `LookupFacts`—automatically from the given list of formulae and from an internal list of "known implications". For looking up $\exists y : P$, we allow a special construct `forsome[y]` to occur in formulae. After a successful look-up we have access to an appropriate binding for y via `this[y]`. This allows for a rather elegant programming style, otherwise not available in *Mathematica*, as shown in the following statement taken from the group simplifier:

```
If[LookupFacts[isIdentity[forsome[n], op, G]],
  id = this[n];
  ... (* cancel op[G,id,_] and op[G,_,id] from expr *)
```

This mechanism is used extensively in our formalization of group theory, it is obviously applicable in other contexts as well.

3.2.2 Where We Use Look-up

As mentioned earlier, the look-up mechanism was originally designed just for checking side-conditions during computations and simplifications, but it quickly turned out that it has a much broader spectrum of applicability. In the current implementation, we exploit `LookupFacts` in the following places:

- In the group simplifiers, see Section 4 for testing the group properties and for actually finding the group operation, the identity, and the inverse,
- In the simplification of sequents, see Section 2.3. In usual sequent calculus, one would delete a formula A in the consequent if A appears in the antecedent. We delete A, if we can look up A in the antecedent! Similarly, we delete parts of conjunctions in the consequent as soon as they can be looked up, and so forth.
- Custom formula simplifications, see Section 2.3, are performed in a context where Γ contains the current antecedent, i.e. any look-up done in a custom simplification rule is relative to the knowledge in the antecedent!
- When translating expressions from Analytica to *Mathematica* we check side-conditions by look-up, e.g. non-zero denominator.

We also refer to the examples in Section 5, which will show the wide range of applicability.

4 Special Techniques

In addition to the logical proof rules, which follow standard sequent calculus, Analytica V contains various specialized rules for special proof situations and/or special theories.

4.1 The Group Simplifiers

We provide a *general-purpose simplifier*, which inspects all terms and simplifies them using associativity, commutativity, identity, and inverses, where checking the properties is done by our look-up mechanism. In addition to that, we also have the possibility to get *specific simplifiers* generated automatically—depending on which properties the structure actually satisfies! As an example, if the knowledge base tells us that G with "+" and "0" forms a monoid, then we can *generate* a simplifier for G that uses associativity of "+" and "0" as the identity w.r.t. "+"—without permanently testing for these properties—and use that simplifier for terms constructed by "+" on G.

4.2 Symmetry Analysis

We analyze conjunctions in the goal and disjunctions in the assumptions to detect symmetries. As a simple example, consider:

$$\texttt{seq}[\{a > 0, b > 0, x > a + b\}, \{x > a \wedge x > b\}]$$

Since the sequent is symmetric about a and b, we can reduce the conjunction in the goal:

$$\texttt{seq}[\{a > 0, b > 0, x > a + b\}, \{x > a\}]$$

More formally, in a sequent $seq[l, r]$, we reduce conjunction $A \wedge B$ in the goal (or disjunction $A \vee B$ in the assumptions), if there exists a permutation f of the parameters in the sequent such that:

$$l[f] =_\alpha l, r[f] =_\alpha r, A[f] =_\alpha B$$

where $=_\alpha$ denotes equality up to alpha conversion and some simple normalizations relating to commutative operators and predicates that are symmetric about their arguments.

We use *Mathematica*'s `Solve`, `Reduce` and `Eliminate` method, see [Wol02], to obtain suggested instantiations of variables in systems of equations over the reals or complexes. In deciding which sets of variables to eliminate when solving a given system, we make use of the symmetry machinery: if the sequent in symmetric about variables a and b, we either try to eliminate both a and b or neither of the two.

4.3 Heuristic Instantiation

In cases where unification and equation solving are insufficient we use an instantiation heuristic. As an example, consider the following sequent:

$$\texttt{seq}[\{y \in \mathbb{Z}, \forall_{x\in\mathbb{Z}} f(x) < f(x+1)\}, \{f(y) < f(y+2)\}]$$

Since $f(y)$ is a term present in the sequent, it is natural to instantiate the universal quantifier on the left with y. To this end we introduce a general rule that considers instantiations of universal quantifiers on the left and existential quantifiers on the right with terms in the sequent, according to built-in heuristics that evaluate the formula resulting from an instantiation.

4.4 Mathematica *Built-Ins*

Needless to say, Analytica V relies heavily on the symbolic capabilities of *Mathematica* for simplification and for decision procedures. The operations used most frequently in the current proofs are the following.

- `FindInstance` is a decision procedure that applies various techniques for systems real, complex, and integer valued equations and inequalities. We use `FindInstance` for refutations after collecting suitable equations and inequalities in the antecedent and consequent, see section 5.2 below for an application.
- `GroebnerBasis` is an implementation of Buchberger's Gröbner basis algorithm. We use the algorithm to derive contradictions among sets of polynomial equations. This method is employed in the proof of associativity of the group law for rational points on an elliptic curve.
- `FullSimplify` is a general purpose simplifier which tries a wide range of transformations. It is employed frequently to simplify algebraic expressions in the consequent, using information from the antecedent as assumptions.

Since the exact theory, in which some of those *Mathematica*-algorithms are valid, is in some cases not spelled out, it is impossible to formally specify the theory w.r.t. which certain Analytica-proofs are valid. Some computer-algebra-based proof steps, however, are provenly correct, and in these cases the underlying theory is well-known. As an example, take the reduction of proving the unsolvability of a system of polynomial equations to checking its reduced Gröbner basis to be $\{1\}$, which is proven to be true in the theory of multivariate polynomials over some field. Of course, we rely/depend on the correct implementation of the Gröbner basis algorithm in *Mathematica*, but this does not differ from using one's own implementation. In other cases the situation is unfortunately much less clear, e.g. when using `FullSimplify`, we need to trust that what *Mathematica* returns as a simplified expression is equal/equivalent to the original expression (w.r.t. some theory) as claimed by the *Mathematica*-manual. At least, as mentioned in Section 3.2.2, when converting expressions from Analytica to *Mathematica* we explicitly check certain side-conditions, which are assumed by *Mathematica* implicitly.

4.5 The Handling of Definitions

We support both *explicit definitions* of predicates and functions and *implicit function definitions*. As a special language construct, a *case distinction* may be used define functions or predicates by cases. Accordingly, if functions or predicates defined by cases are present in a proof, we provide proof rules to split the proof into the respective cases. Moreover, we employ predicate definitions when building up the look-up table, see Section 3.2.1.

5 Examples

5.1 Group Characters

The proof of Dirichlet's Theorem refers to the concept of number-theoretic group characters. For an abelian multiplicative group G a *group character* is a homomorphism from $(G, \cdot)$ to $(\mathbb{C}^*, \cdot)$, where $\mathbb{C}^*$ denotes the multiplicative subgroup of the complex numbers. The set $GC[G]$ of group characters on G has a natural multiplication, namely, $(a \cdot b)(x) = a(x)b(x)$.

Theorem 5.1 *$GC[G]$ is an abelian multiplicative group with the identity character $\mathbf{1}(x) = 1$ and the inverse character*[10] *$reciprocal(a)(x) = a(x)^{-1}$.*

The prover configuration for a completely automated proof of this theorem is as follows: The rulebase contains a rule for expanding definitions in the consequent, a rule for verifying equations in the consequent, the logical rules for And-Right and Or-Left as described in Section 2.3, and the standard normalization rule. For building up the look-up table as described in Section 3.2.1, we use all algebraic definitions known to Analytica, which define semigroups, monoids, groups,..., homomorphisms, group characters hierarchically. We will show parts related to proving the inverse-property in more detail. For the annotated presentation of the automated proof below we use abbreviations for function- and predicate symbols introduced in the formalization of the theorem: $\mathrm{AMG}[G, \mathbf{1}, \mathbf{i}]$ formalizes "G is an abelian multiplicative group with identity $\mathbf{1}$ and inverse function $\mathbf{i}$".

Proof.

$\forall_{\mathrm{inv}} \forall_{\mathrm{id}} \forall_G \ \mathrm{AMG}[G, \mathrm{id}, \mathrm{inv}] \Rightarrow \mathrm{AMG}[\mathrm{GC}[G], \mathbf{1}, \mathrm{reciprocal}]$	Theorem: If G is an abelian multiplicative group then also the group characters on G form an abelian multiplicative group.

[10] Note the currying in the definition of both multiplication and inverse of group characters, see Section 2.1.

We have to show:

$$\text{AMG}[\text{GC}[G_2], \mathbf{1}, \text{reciprocal}]$$

$$\vdots$$

We have to show:

$$\text{isInverse}[\\ \quad \text{reciprocal}, \text{times}, \mathbf{1}, \text{GC}[G_2]]$$

We expand definitions in the goal, thus, the new proof goal is

$$\forall_{a \in \text{GC}[G_2]} a^{-1} \in \text{GC}[G_2] \wedge$$

$$\forall_{a \in \text{GC}[G_2]} a *_{\text{GC}[G_2]} a^{-1} =_{\text{GC}[G_2]} \mathbf{1}$$

$$\vdots$$

$$(a_1^{-1})[x_1] *_{\mathbb{C}^*} (a_1^{-1})[y_1] =_{\mathbb{C}^*}$$

$$(a_1^{-1})[x_1 *_{G_2} y_1].$$

Function definitions expand inside the goal, so we get

$$(a_1[x_1])^{-1} *_{\mathbb{C}^*} (a_1[y_1])^{-1} =_{\mathbb{C}^*}$$

$$(a_1[x_1 *_{G_2} y_1])^{-1}.$$

Custom rewrite rules apply in order to simplify the sequent.
We are left with

$$(a_1[x_1])^{-1} *_{\mathbb{C}^*} (a_1[y_1])^{-1} =_{\mathbb{C}^*}$$

$$(a_1[x_1] *_{\mathbb{C}^*} a_1[y_1])^{-1}.$$

During conversion from an Analytica expression to its corresponding *Mathematica* expression we need to check

$$a_1[x_1] *_{\mathbb{C}^*} a_1[y_1] \neq 0$$

but this follows easily from

Normalization skolemizes the universal quantifiers and splits the implication.
Stepwise expanding definitions. Split conjunction in consequent. Closure, associativity, identity-property all proved. Finally, we expand the definition of being an inverse. Note, that all operations carry information about their domain, we denote it as a subscript. Now, the conjunction will be split, and we concentrate on the first part, i.e. that the inverse character in fact *is a group character* on G_2. Again, skolemization and expansion of $a_1 \in \text{GC}[G_2]$, which means, a_1 must be a homomorphism from G_2 into $\mathbb{C}^*$.

Several properties have to be proven: a_1^{-1} must first of all be a mapping, etc. we skip forward to the interesting property that the inverse character a_1^{-1} has the homomorphism property. Note that, in a previous simplification attempt, the left and right side of the equality have been swapped. Now the *definition of the inverse character* applies on both sides, essentially pulling the $^{-1}$ outside. On the right side we can now use the property that a_1 is a homomorphism. Of course, the simplification rule for homomorphisms is implemented in such a way that it *verifies* that a_1 actually *is a homomorphism*, which is easy by look-up since we know a_1 to be a group character on G_2. We are left with an equality on $\mathbb{C}$, which we simply ship to *Mathematica*. During conversion from Analytica to *Mathematica* we check side-conditions, notably the expressions to be inverted must be unequal zero! In this case, we look up an expression of the form $x * y \neq 0$, and the look-up proceeds by looking up $x * y \in \text{forsome}[D]^*$. Now, $x * y \in D$ will look up $\text{isClosed}[*, D] \wedge x \in D \wedge y \in D$,

$$\mathrm{AMG}[\mathbb{C}^*, 1, \mathrm{reciprocal}],$$
$$a_1 \in \mathrm{GC}[G_2],$$
$$x_1 \in G_2,$$
$$y_1 \in G_2.$$

The equality can be proved by simplification.

$\vdots$

where the search will branch. Looking up isClosed$[*, D]$ will go through the entire algebraic hierarchy through semigroup, monoid, etc. finally succeeding through built-in knowledge from the prover configuaration that $\mathbb{C}^*$ is an abelian multiplicative group. $a_1[x_1], a_1[y_1] \in \mathbb{C}^*$ it will find since a_1 is a group character on G_2 and $x_1, y_1 \in G_2$.

The remaining parts of the proof use the same techniques as already illustrated. □

The issue worth mentioning is that we need not expand definitions in the antecedent but still have certain hidden knowledge inferable from the antecedent available in the proof. For example, we certainly do not wish to expand the fact that $\mathbb{C}^*$ is an abelian multiplicative group down to the property that $\mathbb{C}^*$ is closed under multiplication. This becomes even more important when we think about dealing with large knowledge bases. Analytica can also prove a more general form of this theorem: the set of homomorphism (with multiplication like group characters) between two abelian groups forms a group. The proof goes along the same lines, except that we cannot send equalities to *Mathematica* to be verified. Here, the group simplifiers described in Section 4 come into play, and they have no problems with the group simplifications needed in these examples.

5.2 The Weak Mordell-Weil Theorem

The Weak Mordell-Weil Theorem states that, for any elliptic curve $E(\mathbb{Q})$, the quotient group $E(\mathbb{Q})/2E(\mathbb{Q})$ is finite. We consider the proof of this fact for elliptic curves of the form $y^2 = (x-a)(x-b)(x-c)$ where a, b, c are distinct rational numbers. The proof, as given in [KKS00], proceeds as follows.

We define a mapping $\delta : E(\mathbb{Q}) \to \mathbb{Q}^*/(\mathbb{Q}^*)^2 \times \mathbb{Q}^*/(\mathbb{Q}^*)^2 \times \mathbb{Q}^*/(\mathbb{Q}^*)^2$ by:

$$\delta(P) = \begin{cases} (1,1,1) & \text{if } P = O, \\ ((a-b)(a-c), a-b, a-c) & \text{if } P = (a,0), \\ (b-a, (b-a)(b-c), b-c) & \text{if } P = (b,0), \\ (c-a, c-b, (c-a)(c-b)) & \text{if } P = (c,0), \\ (x-a, x-b, x-c) & \text{otherwise, } P = (x,y). \end{cases}$$

The proof of the theorem comprises three parts:

(i) δ is a homomorphism.

(ii) The kernel of δ is $2E(\mathbb{Q})$.

(iii) The image of delta is contained in the finite subgroup generated by the prime factors of $a-b$, $b-c$, $c-a$, -1.

By the first isomorphism theorem, we then have that $E(\mathbb{Q})/2E(\mathbb{Q})$ is isomorphic to a finite group. Our status in automating each of these pieces is as follows:

(i) This part has been automated in the most general case, i.e. for points in general positions. Missing for full automation of the other cases are some simplification mechanisms.

(ii) For this part, we consider a decomposition of the multiplication-by-two map into functions g, h, such that $h \circ g$ is the map. From this decomposition it is easily seen that the image of the map is contained in the kernel of δ. The reverse containment is implied by the surjectivity of g from $E(Q)$ onto a special domain, and the surjectivity of h from that domain onto the kernel of δ, both of which we have automated.

(iii) We have fully automated this part. Analytica handles the case-splitting correctly and takes care of each case.

Here we present a shortened and edited proof of the third part, as generated by Analytica. Except for pretty-printing we have retained most of the Analytica syntax. Thus, image$[f, D]$ denotes the image of f over domain D. $\delta[a, b, c]$ refers to the map δ associated with the curve parameters a, b and c. elemGen $[s, *, G, P]$ means that element s is generated from P in group $\langle G, *\rangle$. We write pV$[p, q]$ for the p-adic valuation of q.

Theorem 5.2

$$\forall_{a\in\mathbb{Q}}\left(\forall_{b\in\mathbb{Q}}\left(\forall_{c\in\mathbb{Q}} image\left[\delta[a,b,c], E(Q)\right] \subseteq G \times G \times G\right)\right)$$

where

$$G = \left\{\, s \;\middle|\; elemGen\left[s, times, \left(\mathbb{Q}/\mathbb{Q}^2\right) \times \left(\mathbb{Q}/\mathbb{Q}^2\right) \times \left(\mathbb{Q}/\mathbb{Q}^2\right), P\right] \right\}$$

and

$$P = \{p\mathbb{Q}^2 \;\big|\; p =_{\mathbb{Z}} -1 \vee pV[p, a -_{\mathbb{Q}} b] \neq_{\mathbb{Z}} 0 \vee \\ pV[p, b -_{\mathbb{Q}} c] \neq_{\mathbb{Z}} 0 \vee pV[p, c -_{\mathbb{Q}} a] \neq_{\mathbb{Z}} 0\}$$

Proof. After expanding the definitions the first interesting event is the following new formula in the antecedent.

$$\mathrm{s} =_{(\mathbb{Q}/\mathbb{Q}^2)^3} \mathrm{vector}\,[1,1,1] \vee \mathrm{s} \in \{\delta(x,y) | x, y \in \mathbb{Q}, y^2 =_{\mathbb{Q}} (x-a)(x-b)(x-c)\}$$

The case where $s = (1, 1, 1)$ is easily taken care of. Next we consider the image of δ. The case distinction mechanism first tackles $x =_{\mathbb{Q}} a$. The crucial event associated with this case is the following formula in the consequent:

$$\text{elemGen}\left[(a -_{\mathbb{Q}} b) *_{\mathbb{Q}} (c -_{\mathbb{Q}} a), *, \mathbb{Q}/\mathbb{Q}^2, P\right] \wedge \text{elemGen}\left[a -_{\mathbb{Q}} b, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$
$$\wedge\ \text{elemGen}\left[c -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

This case is easily handled since $(a-b)(a-c)$, $a-b$, $a-c$ are trivially generated by P. The cases $x =_{\mathbb{Q}} b$ and $x =_{\mathbb{Q}} c$ are similar. The remaining generic case has additional assumptions $x \neq_{\mathbb{Q}} a, x \neq_{\mathbb{Q}} b, x \neq_{\mathbb{Q}} c$. The key event here is

$$\text{elemGen}\left[x -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right] \wedge \text{elemGen}\left[x -_{\mathbb{Q}} b, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$
$$\wedge\ \text{elemGen}\left[x -_{\mathbb{Q}} c, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

Exploiting symmetry, the prover now establishes only the assertion

$$\text{elemGen}\left[x -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

Using an auxiliary lemma[11] and backchaining the proof goal is now translated into an assertion about p-adic valuations: from assumptions

$$p \in \text{Primes}, \text{pV}\left[p, x -_{\mathbb{Q}} a\right] \notin 2\mathbb{Z}.$$

we need to establish the goals

$$\text{pV}\left[R, a -_{\mathbb{Q}} b\right] \neq_{\mathbb{Z}} 0, \text{pV}\left[R, b -_{\mathbb{Q}} c\right] \neq_{\mathbb{Z}} 0, \text{pV}\left[R, c -_{\mathbb{Q}} a\right] \neq_{\mathbb{Z}} 0.$$

To this end we use properties of p-adic valuations stored in the knowledge base:

$$\text{pV}\left[p, s -_{\mathbb{Q}} t\right] \geq_{\mathbb{Q}} \min[\mathbb{Z}, \text{pV}[p,s], \text{pV}[p,t]]$$
$$\text{pV}[p,s] \neq_{\mathbb{Q}} \text{pV}[p,t] \Rightarrow \text{pV}\left[p, s -_{\mathbb{Q}} t\right] =_{\mathbb{Q}} \min[\mathbb{Z}, \text{pV}[p,s], \text{pV}[p,t]]$$

By heuristic instantiation we now generate additional assumptions such as

$$\min[\mathbb{Q}, \text{pV}[p, -a+x], \text{pV}[p, -b+x]] \leq \text{pV}[p, a-b].$$

and

$$\text{pV}[p, x-a] =_{\mathbb{Q}} \text{pV}[p, c-a] \vee \text{pV}[p, x-c] =_{\mathbb{Q}} \min[\mathbb{Q}, \text{pV}[p, c-a], \text{pV}[p, x-a]].$$

At this point the prover uses `FindInstance`, a *Mathematica* decision algorithm for the existence of solutions of systems of equations and inequalities, to refute these additional premises and the negated goals.

$\square$

[11] The supplied auxiliary lemma states that if for all primes p such that $\text{pV}[p,t]$ is odd, we have $p \in S$, and $-1 \in S$, then S generates $t\mathbb{Q}^2$ in $\mathbb{Q}^*/(\mathbb{Q}^*)^2$

6 Conclusion

The work reported in this paper is work in progress towards a fully automated proof of the Mordell-Weil Theorem and a theorem of Dirichlet. The proof search, the look-up mechanism, and the handling of definitions are completely general, however, and their interplay behaved very promising in the examples done up to now. In particular, the fully automated setup of a proof by case distinction based on definitions by cases will be applied to proving the associativity of the group law on elliptic curves, a symbolic computation proof, which has probably newer been done up to now due to its many cases to distinguish.

References

[BCZ98] A. Bauer, E. Clarke, and X. Zhao. Analytica — an Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[CKOS03] E. Clarke, M. Kohlhase, J. Ouaknine, and K. Sutner. Analytica 2. In *Calculemus, Rome*, 2003.

[CZ92] Edmund Clarke and Xudong Zhao. Combining symbolic computation and theorem proving: Some problems of ramanujan. In D. Kapur, editor, *Proceedings the 11th Conference on Automated Deduction*, volume 607 of *LNCS*, pages 66–78, Saratoga Spings, NY, USA, 1992. Springer Verlag.

[KKS00] Kazuya Kato, Nobushige Kurokawa, and Takeshi Saito. *Number Theory 1: Fermat's Dream.* Translations of Mathematical Monographs. American Mathematical Society, 2000. ISBN 0-821-80863-X.

[Sil86] Joseph H. Silverman. *The Arithmetic of Elliptic Curves.* Graduate Texts in Mathematics. Springer-Verlag, 1986. ISBN 0-387-96203-4.

[Win99] W. Windsteiger. Building Up Hierarchical Mathematical Domains Using Functors in Theorema. In A. Armando and T. Jebelean, editors, *Electronic Notes in Theoretical Computer Science*, volume 23-3, pages 83–102. Elsevier, 1999. Calculemus 99 Workshop, Trento, Italy.

[Wol02] S. Wolfram. *The Mathematica Book.* Cambridge University Press, 2002.