Mizar Correctness Proofs of Generic Fraction Field Arithmetic *

Christoph Schwarzweller
Wilhelm-Schickard-Institute for Computer Science
Sand 13, D-72076 Tübingen
schwarzw@informatik.uni-tuebingen.de

24th July 1998

Abstract

We propose the MIZAR system as a theorem prover capable of verifying generic algebraic algorithms on an appropriate abstract level. The main advantage of the MIZAR theorem prover is its special proof script language that enables textbook style presentation of proofs, hence allowing proofs in the language of algebra.

Using MIZAR we were able to give a rigorous machine assisted correctness proof of a generic version of the Brown/Henrici arithmetic in the field of fractions over arbitrary gcd domains.

1 Introduction

Over the last several years generic programming has received more and more attention. Many programming languages nowadays include generic concepts like polymorphism in functional programming languages, overloading or templates in C++; or they are even completely designed as a generic language like SUCHTHAT [7]. Also generic libraries have been developed like the ADA Generic Library or the STL. On the other hand the widespread use of generic concepts more and more entails the need for a thorough formal machine assisted verification of generic algorithms — especially to improve the reliability of generic libraries. This paper

*This paper is based on the author's Ph.D-thesis[8], supervised by Rüdiger Loos.

deals with the verification of generic algorithms in the field of computer algebra.

SuchThat is a programming language that enables generic programming in the field of computer algebra. SuchThat is a procedural language and can be seen as a successor of Aldes [4] from which it adopted its statement parts. The main feature of SuchThat is the possibility to express the specification of an algorithm (and hence the minimal conditions under which the algorithm works) in the language itself. To achieve this, SuchThat contains declarations that enable the user to introduce parametrized and attributed algebraic structures. Subsequently, algorithms are written based on these structures.

We consider the algorithm of Brown and Henrici concerning addition of fractions over gcd domains as an example. Let I be an integral domain, and let Q be the set of fractions over I. Based on algorithms for arithmetic operations in I one obtains algorithms for arithmetics in Q. To be able to choose a unique representative from each equivalence class of Q, we assume that I is a gcd domain; that is, an integral domain in which for any two elements a greatest common divisor exists. We also assume that there is a constructor `fract` to build a fraction out of elements of `I` and selectors `num` and `denom` that decompose a fraction into numerator and denominator respectively.

The algorithm accepts normalized fractions as input, returning the result again as a normalized fraction. The point is that the normalized result is achieved not by executing ordinary addition of fractions followed by a normalization step, but by integrated greatest common divisor computations. First, this allows singling out trivial cases and, more important second, taking advantage of the normalization of the inputs to achieve normalization of the output by, in general, cheaper gcd-computations. In SuchThat the algorithm is written as follows.

```
"BrHenAdd.sth" 2 ≡
   global: let I be gcdDomain;
           let Q be Fractions of I;
           let Amp be multiplicative AmpleSet of I.

   Algorithm: t := BHADD(r,s)
       Input: r,s ∈ Q such that r,s is_normalized_wrt Amp.
      Output: t ∈ Q such that t~r+s & t is_normalized_wrt Amp.
       Local: let 0,1,r1,r2,s1,s2,d,e,r2',s2',t1,t2,t1',t2' ∈ I;
              let 0 ∈ Q;
```

```
(1) [r = 0 or s = 0]
    if r = 0 then {t := s; return};
    if s = 0 then {t := r; return}.
(2) [get numerators and denominators]
    r1 := num(r); r2 := denom(r); s1 := num(s); s2 := denom(s).
(3) [r and s in I]
    if (r2 = 1 and s2 = 1) then {t := fract(r1+s1,1); return}.
(4) [r or s in I]
    if r2 = 1 then {t := fract(r1*s2+s1,s2); return};
    if s2 = 1 then {t := fract(s1*r2+r1,r2); return}.
(5) [general case]
    d := gcd(r2,s2);
    if d = 1 then {t := fract(r1*s2+r2*s1,r2*s2); return};
    r2' := r2/d; s2' := s2/d;
    t1 := r1*s2'+s1*r2'; t2 := r2*s2';
    if t1 = 0 then {t := 0; return};
    e := gcd(t1,d); t1' := t1/e; t2' := t2/e;
    t:= fract(t1',t2'). □
```

◇

File defined by parts 2, 3, 11.

For completeness we also give the SuchThat specifications of the necessary subalgorithms. Note that for the verification of Brown/Henrici addition we only need these so-called prototypes rather than the full subalgorithms. Consequently Brown/Henrici addition will be correct for every set of subalgorithms fulfilling these specifications regardless of how these algorithms are written in detail.

"BrHenAdd.sth" 3 ≡

```
Algorithm: r1 := num(r)
Input:     r ∈ Q.
Output:    r1 ∈ I such that r1 = num(r). □

Algorithm: r2 := denom(r)
Input:     r ∈ Q.
Output:    r2 ∈ I such that r2 ≠ 0 & r2 = denom(r). □

Algorithm: r := fract(r1,r2)
Input:     r1,r2 ∈ I such that r2 ≠ 0.
Output:    r ∈ Q such that r = fract(r1,r2). □
```

```
Algorithm: d := /(r1,r2)
Input:     r1,r2 ∈ I such that r2 ≠ 0 & r2 divides r1.
Output:    d ∈ I such that d = r1/r2. □

Algorithm: c := gcd(a,b)
Input:     a,b ∈ I.
Output:    c ∈ I such that c ∈ Amp & c = gcd(a,b). □
```
◇
File defined by parts 2, 3, 11.

SuchThat is currently a preprocessor to C++. The static semantic checks of the SuchThat translator include extended typechecking, overload resolution and controlled instantiation of structure parameters. SuchThat programs can be instantiated with special domains in the usual way. For example the Brown/Henrici algorithm can be instantiated with the integers, polynomial rings or the Gaussian integers.

The correctness of this algorithm depends on fundamental properties of greatest common divisors. In the following we will show how to prove them (and the correctness of the algorithm) rigorously with machine assistance.

2 The Mizar System

Mizar [6] is a theorem prover based on natural deduction. Starting from the axioms of set theory[1], up to now about 20,000 theorems from such different fields of mathematics as topology, algebra, category theory and many others have been proven and stored in a library. From our point of view one of the main contribution of the Mizar system is its special proof script language. This language is declarative and associates the natural deduction steps with English constructs, thus allowing to write proofs close to textbook style.

In addition Mizar includes a kind of mathematical type system: The user can define so-called *modes*, that is mathematical structures and objects he wants to argue about (for example `integral domain` or `domRing`, as it is called in Mizar, is such a mode). Consequently, we can write

```
let I be domRing;
let a be Element of the carrier of I;
```

[1]To be more precise, Mizar uses the axioms of a variant of ZFC set theory due to Tarski.

Now domRing is defined as a commutative ring which satisfies the following predicate, in MIZAR called *attribute*.

```
definition
let R be comRing;
attr R is domRing-like means
  for x, y being Element of the carrier of R holds
  x * y = 0.R implies x = 0.R or y = 0.R;
end;
```

As a consequence it inherits all properties of (commutative) rings — especially, all already proven theorems concerning rings are applicable to I. So, if we already have proven the following theorem

```
T : for R being Ring
  for x being Element of R holds
  x * 0.R = 0.R;
```

where T is a label, we only have to write

```
a * 0.I = 0.I by T;
```

to prove $a \cdot 0 = 0$ in an integral domain I.

But this is just the platform we need to reason about generic algebraic algorithms: We argue in abstract algebraic domains, we use only formal parameters that hold for every possible instantiation. Hence we prove the algorithms correct on the appropriate abstract algebraic level.

Due to its natural proof script language, MIZAR is well suitable not only to formalize mathematics, but also for scientists writing generic (algebraic) algorithms: They can prove the correctness of their algorithms in MIZAR in almost the same way they would prove them without machine assistance and need not in addition to go deeply into a proof logic or the strategies of a special theorem prover.

In the following we give a short introduction to the MIZAR language by using parts of our MIZAR article GCD.miz as illustrations.[1] Each MIZAR article starts with an environment which introduces mathematical concepts one wants to use in

[1]For more information on the MIZAR language and the MIZAR system see the MIZAR home page at http://mizar.org/.

the rest of the article. This is done by referencing other MIZAR articles in which the desired concepts have been defined. In our case:

"GCD.miz" 6a ≡

```
environ
  vocabulary
  BOOLE,VECTSP_1,VECTSP_2,REAL_1,LINALG_1,SFAMILY,GCD;
  notation
  TARSKI,BOOLE,STRUCT_0,RLVECT_1,SETFAM_1,VECTSP_1,VECTSP_2;
  constructors
  ALGSTR_1;
  theorems
  TARSKI,BOOLE,WELLORD2,SUBSET_1,ENUMSET1,VECTSP_1,VECTSP_2;
```

◇

File defined by parts 6ab, 7.

The second part of a MIZAR article is called the text proper. It includes new mathematical definitions and theorems as well as proofs for those. Based on the vocabulary introduced in the environment one can define new mathematical concepts in a very natural way. For example, to introduce the concept of divisibility in integral domains one can write

"GCD.miz" 6b ≡

```
reserve I for domRing;
reserve a,b,c for Element of the carrier of I;

definition
let I; let a,b,c;
pred a divides b means :Def1:
 ex c being Element of the carrier of I st b = a*c;
end;

definition
let I; let a,b,c;
pred a is_associated_to b means :Def2:
 a divides b & b divides a;
antonym a is_not_associated_to b;
end;
```

◇

File defined by parts 6ab, 7.

In addition one can define functions in MIZAR, for example the division function over integral domains. In contrast to introducing predicates this requires a correctness proof, that is an existence and a uniqueness proof. @proof is a nice construct that enables the user to postpone required proofs and to concentrate on the implications of the definitions.[1]

```
"GCD.miz" 7 ≡
   definition
   let I; let a,b,c;
   assume d: b divides a & b <> 0.I;
     func a/b -> Element of the carrier of I means :Def5:
     it*b = a;   :: it stands for the value of a/b.
   correctness @proof end;
   end;
```

◇

File defined by parts 6ab, 7.

Unfortunately, the algebraic domains necessary to show correctness of Brown/ Henrici arithmetics, namely gcd domains and fractions, were not included in the MIZAR library so far. So we first had to prove about 40 theorems about divisibility, gcd domains and fractions before we could start the entire correctness proof.

3 Verifying Generic Algebraic Algorithms

To prove correct generic algebraic algorithms, or to be more precise SUCHTHAT algorithms, we first decompose the given algorithm using the Hoare calculus. We consider the algorithm with its input/output specification as a Hoare triple, on which we apply Hoare's rules in a backward manner. The key is that this allows us to eliminate the whole program code out of the given triple, that is, we end up with pure algebraic theorems. So we get a set of algebraic theorems[1] implying the correctness of this algorithm on the appropriate abstract level independent of any particular instantiation.

We implemented in SCHEME a verification condition generator following this approach (see [8]). Of course this generator needs user support to compute a complete verification condition set, for example loop invariants must be provided

[1] Of course the MIZAR Library Committee will not accept an article from which not all @proof's are removed.

[1] In fact this is a so-called verification condition set, see for example [2].

by the user. For the Brown/Henrici addition algorithm the generator constructs 15 theorems, in this case all without user interaction. We give an example theorem which is already transformed into the MIZAR language: It states that `t` computed in step `(5)` of the algorithm, where `d = gcd(r2,s2) = 1`, indeed is normalized and equivalent to `r+s`. The MIZAR proof of this theorem can be found in the appendix.

"BrHenArith.miz" 8a ≡

```
theorem
(Amp is_multiplicative &
 r is_normalized_wrt Amp & s is_normalized_wrt Amp &
 not(r = 0.Q) & not(s = 0.Q) &
 r1 = num(r) & r2 = denom(r) & r2 <> 0.G & r2 <> 1.G &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G & s2 <> 1.G &
 not(r2 = 1.G & s2 = 1.G) & d ∈ Amp & d = 1.G &
 d = gcd(r2,s2,Amp) & d = 1.G &
 t = fract(r1*s2+r2*s1,r2*s2))
implies (t ~ r+s & t is_normalized_wrt Amp)
⟨proof of the theorem 15, ... ⟩
```

◇

File defined by parts 8ab, 9ab, 10, 12.

To prove these automatically constructed theorems we also have to introduce the concepts used in the algorithm; that is in addition to the algebraic domains — gcd domains and fractions — we must define in MIZAR ample sets (or sets of representatives) over integral domains and the predicates `~` and `is_normalized_wrt` as well as functions representing the subalgorithms of section one. Again we only give some examples, mainly to show how to define mathematical objects in the MIZAR language.

"BrHenArith.miz" 8b ≡

```
definition
let I be domRing;
mode AmpSet of I -> non empty Subset of the carrier of I means
  (for a being Element of the carrier of I
   ex z being Element of it st z is_associated_to a) &
  (for x,y being Element of it holds
   x <> y implies x is_not_associated_to y);
existence @proof end;
end;
```

```
  definition
  let I be domRing;
  attr I is gcd-like means
    (for x,y being Element of the carrier of I
     ex z being Element of the carrier of I st
     z divides x &
     z divides y &
     (for zz being Element of the carrier of I
      st (zz divides x & zz divides y) holds zz divides z));
  end;
```

◇

File defined by parts 8ab, 9ab, 10, 12.

A gcd domain is simply an integral domains whith the just defined attribute `gcd-like`. But before we can introduce mode `gcdDomain` in this way we have to prove the existence of such an object[1] in a so-called cluster definition.

"BrHenArith.miz" 9a ≡

```
  definition
    cluster gcd-like domRing;
  existence @proof end;
  end;

  definition
    mode gcdDomain is gcd-like domRing;
  end;
```

◇

File defined by parts 8ab, 9ab, 10, 12.

Now having introduced gcd domains in MIZAR it is easy to define the remaining concepts e.g. the predicate `is_normalized_wrt`.

"BrHenArith.miz" 9b ≡

```
  definition
  let G be gcdDomain;
  let u be Fraction of G;
  let Amp be AmpleSet of G;
```

[1] MIZAR does not allow empty modes.

```
   pred u is_normalized_wrt Amp means :Def20:
     gcd(num(u),denom(u),Amp) = 1.G &
     denom(u) ∈ Amp;
   end;
```
◇

File defined by parts 8ab, 9ab, 10, 12.

Proving the verification conditions requires as the main part the proof of the so-called theorem of Brown and Henrici [1], which shows how the sum of two normalized fractions can be computed in a normalized form with the reduction to lowest terms interleaved with the computation of the numerator and denominator of the sum. `r1` resp. `s1` are the numerators and `r2` resp. `s2` the denominators of the occurring fractions.

"BrHenArith.miz" 10 ≡
```
   theorem
   for Amp being AmpleSet of I
   for r1,r2,s1,s2 being Element of the carrier of I holds
   (gcd(r1,r2,Amp) = 1.I & gcd(s1,s2,Amp) = 1.I &
    r2 <> 0.I & s2 <> 0.I)
   implies
   gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
       r2*(s2/gcd(r2,s2,Amp)),Amp) =
   gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
       gcd(r2,s2,Amp),Amp)
   @proof end;
```
◇

File defined by parts 8ab, 9ab, 10, 12.

We will not give the MIZAR proof of this theorem here; it can be found in [8]. We only mention that the entire correctness proof (that is the introduction of gcd domains and ample sets is not included) took about 1000 lines of MIZAR code. Note again, that the verification proofs are done on an abstract algebraic level, hence that they are independent of any particular instantiation.

It is obvious that the subtraction algorithm follows by the same token as the addition algorithm. For multiplication Henrici and Brown provide the following algorithm:

"BrHenAdd.sth" 11 ≡

```
   global: let I be gcdDomain;
           let Q be Fractions of I;
           let Amp be multiplicative AmpleSet of I.

   Algorithm: t := BHMULT(r,s)
       Input: r,s ∈ Q such that r,s is_normalized_wrt Amp.
      Output: t ∈ Q such that t~r*s & t is_normalized_wrt Amp.
       Local: let 1,r1,r2,s1,s2,d,e,r1',r2',s1',s2' ∈ I;
              let 0 ∈ Q;

   (1) [r = 0 or s = 0]
       if r = 0 or s = 0 then {t := 0; return}.
   (2) [get numerators and denominators]
       r1 := num(r); r2 := denom(r); s1 := num(s); s2 := denom(s).
   (3) [r and s in I]
       if (r2 = 1 and s2 = 1) then {t := fract(r1*s1,1); return}.
   (4) [r or s in I]
       if r2 = 1 then {d := gcd(r1,s2); r1' := r1/d;
                       t := fract(r1'*s1,s2); return};
       if s2 = 1 then {d:= gcd(s1,r2); r1' := r1/d;
                       t := fract(r1'*s1,r2); return}.
   (5) [general case]
       d := gcd(r2,s2); e := gcd(s1,r2);
       r1' := r1/d; r2' := r2/e;
       s1' := s1/e; s2' := s2/d;
       t:= fract(r1'*s1',r2'*s2'). □
```

◇

File defined by parts 2, 3, 11.

The overall goal is the same as for addition: to make use of the normalization of the inputs to simplify and, in general, speed up the normalization of the output. Division of fractions is reduced to multiplication by the inverse fraction which does not introduce additional complexity.

The correctness proof of this algorithm is similar to the one of the addition algorithm. It is based on the following theorem which is also due to Brown and Henrici:

"BrHenArith.miz" 12 ≡

```
   theorem
   for Amp being AmpleSet of I
   for r1,r2,s1,s2 being Element of the carrier of I holds
   (gcd(r1,r2,Amp) = 1.I & gcd(s1,s2,Amp) = 1.I &
    r2 <> 0.I & s2 <> 0.I)
   implies
   gcd((r1/gcd(r1,s2,Amp))*(s1/gcd(s1,r2,Amp)),
       (r2/gcd(s1,r2,Amp))*(s2/gcd(r1,s2,Amp)),Amp) = 1.I;
   @proof end;
```

◇

File defined by parts 8ab, 9ab, 10, 12.

Our verification condition generator constructs 12 theorems for the multiplication algorithm. Again each theorem is either straightforward or is proved by showing that the theorem of Brown and Henrici is applicable in this case.

To summarize, using Mizar we were able to give complete verification proofs of fraction field arithmetic based on Brown/Henrici's algorithms. Furthermore we did this in a generic way so that correctness is ensured for fractions over arbitrary gcd domains.

4 Conclusions and Further Work

We have presented a new approach to bringing machine assistance into the field of generic programming. Thereby we focused on generic algebraic algorithms and their verification. Using the Mizar system we succeeded in verifying generic versions of Brown/Henrici arithmetic and of Euclid's algorithm on the appropriate algebraic level; thus our proofs are independent of any particular instantiation and hold for all gcd-domains.

The emphasis is on the fact that algebraic proof in Mizar can be directly written in the language of algebra and need not to be transformed into a more or less completely different proof language. In addition we provided a verification condition generator, which computes from a given SuchThat algorithm and user-supplied lemmata the theorems necessary to establish its correctness.

Another point concerns the instantiation of SuchThat algorithms. Consider again the Brown/Henrici addition algorithm. If it is called for example with the

integers, then an obvious correctness condition is that the integers constitute a gcd domain. In addition assume that a (generic) Euclidean algorithm shall be used as a subalgorithm to compute greatest common divisors. The corresponding prototype requires a gcd domain which gives us another condition, namely that Euclidean domains are gcd domains. This kind of correctness conditions arising for the use of generic algebraic algorithms also can be handled with the MIZAR system and form the basis of the algebraic type checks inSUCHTHAT .

MIZAR's original purpose was to bring mathematics — including the necessary proof techniques — onto the computer and to build a library of mathematical knowledge. In fact, so far the library is nothing more than a collection of articles accepted by the MIZAR proof checker: Reusing the knowledge is not supported as well as it needs to be for our purpose. Consequently, to build a verification system for generic algebraic algorithms around the MIZAR system requires some further work. In this context we like to mention two points.

- First of all, we need a tool for searching the MIZAR library. At the beginning of a verification we have to look at which kinds of algebraic domains are already included in the library and which theorems about these domains have been proven.
  We did some experiments using GLIMPSE [5], a powerful indexing and query system: After indexing the files — the MIZAR abstracts in our case — it allows one to look through these files without the need of specifying file names. It enables the user to look for arbitrary keywords, for instance `gcdDomain`, `VectorSpace` or `finite-dimensional`.

- Though the MIZAR system provides a proof script language capable of expressing algebraic structures appropriately, reasoning about these structures sometimes is a bit cumbersome. For example to prove equations in integral domains we had to do each little step using explicitly the domain's axioms. To handle equational reasoning there are well known better methods, for instance *rewriting systems*; for a couple of algebraic domains there even exist canonical rewrite systems. It seems promising to extend MIZAR by such procedural proof techniques.

Writing generic algebraic algorithms is a difficult but rewarding task: One has to look for abstract algebraic domains suitable for the method one wants to imple-

ment; in addition using the constructed generic algorithms with particular instantiations again raises non trivial algebraic questions.

Consequently, writing correct generic algebraic algorithms requires a careful way of dealing with the underlying mathematical structures. We hope that our work is a first step to support a rigorous development of provable correct generic algebraic algorithms.

Acknowledgements: I like to acknowledge the exposition to the subject and the supervision of my thesis by Rüdiger Loos. The hint to the Mizar system came from Sibylle Schupp and David Musser, both at RPI. Finally I have to thank for the hospitality of the Mizar group in Białystok – in particular Andzrej Trybulec – in summer 1998.

References

[1] George E. Collins, *Algebraic Algorithms, chapter two: Arithmetics*, Lecture manuscript, 1974.

[2] Antoni Diller, *Z – An Introduction to FormalMethods*, 2nd ed., Wiley, New York, 1994.

[3] D. Knuth, *Literate Programming*, The Computer Journal 27(2), 97-111, 1984.

[4] Rüdiger Loos and George E. Collins, *Revised Report on the Algorithm Description Language ALDES*, Technical Report WSI-92-14, Wilhelm-Schickard-Institut für Informatik, 1992.

[5] Udi Manber and Burra Gopal, *GLIMPSE — A Tool to Search Entire File Systems*, available by anonymous ftp by `http://glimpse.cs.arizona.edu`.

[6] Piotr Rudnicki, *An Overview of the Mizar Project*, available by anonymous ftp from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, June 1992.

[7] Sibylle Schupp, *Generic Programming — SuchThat one can build an Algebraic Library*, Ph.D. thesis, University of Tübingen, 1996.

[8] Christoph Schwarzweller, *Mizar Verification of Generic Algebraic Algorithms*, Ph.D. thesis, University of Tübingen, 1997.

[9] Andrzej Trybulec, *Some Features of the Mizar Language*, available by anonymous ftp from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, July 1993.

A Proof of the theorem

In this appendix we present the MIZAR proof of the theorem given in section three. We do so for two reasons: First we want to give an impression of how naturally proofs can be formulated in the MIZAR language. Also we want to show how documenting proofs using literate programming tools (see [3]) can make even long proofs readable. The complete correctness proof of the example algorithm can be found in [8].
Note that the use of literate programming allows extraction of the MIZAR proofs from our document giving the corresponding MIZAR article. In general this leads to a proper MIZAR article that can serve as input to the MIZAR checker.

We start the proof establishing that `gcd(r1,r2,Amp) =` and `gcd(s1,s2,Amp)` both equal `1.G`, which follows from the definition of `is_normalized_wrt`.

⟨proof of the theorem 15⟩ ≡

```
proof
M: now assume
H0: Amp is_multiplicative &
    r is_normalized_wrt Amp & s is_normalized_wrt Amp &
    r1 = num(r) & r2 = denom(r) & r2 <> 0.G &
    s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
    d = gcd(r2,s2,Amp) & d = 1.G &
    t = fract(r1*s2+r2*s1,r2*s2);
H3: r2*s2 <> 0.G by H0,VECTSP_2:15;
H1: denom(t) = r2*s2 by H0,H3,F1;
H2: num(t) = r1*s2+r2*s1 by H0,H3,F1;
H4: gcd(r1,r2,Amp) = 1.G & gcd(s1,s2,Amp) = 1.G
    by H0,Def73;
```

◇

Definition defined by parts 15, 16ab, 17.
Definition referenced in part 8a.

To show `gcd(num(t),denom(t),Amp) = 1.G` we apply the theorem of Brown and Henrici — which is stated as theorem GCD:40. This is done by extending the term `gcd(r1*s2+r2*s1,r2*s2,Amp)` — which in fact is nothing else than the required `gcd(num(t),denom(t),Amp)` — to the form the theorem requires. After this application the assumption `gcd(r2,s2,Amp) = 1.G` allows us to infer that the original term also equals `1.G`.

⟨proof of the theorem 16a⟩ ≡

```
H5:   gcd(r1*s2+r2*s1,r2*s2,Amp)
    = gcd(r1*(s2/1.G)+r2*s1,r2*s2,Amp)           by GCD:10
   .= gcd(r1*(s2/1.G)+s1*(r2/1.G),r2*s2,Amp)     by GCD:10
   .= gcd(r1*(s2/1.G)+s1*(r2/1.G),
          r2*(s2/1.G),Amp)                       by GCD:10
   .= gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
          r2*(s2/gcd(r2,s2,Amp)),Amp)                by H0
   .= gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
          gcd(r2,s2,Amp),Amp)              by GCD:40,H4,H0
   .= 1.G                                     by H0,GCD:32;
```

◇

Definition defined by parts 15, 16ab, 17.
Definition referenced in part 8a.

Next we show that `denom(t) = r2*s2` is an element of the ample set `Amp`. This follows from the assumption that `r` and `s` are normalized fractions. Note that we need `Amp` to be multiplicative to conclude `r2*s2 ∈ Amp` at level `H6`.

⟨proof of the theorem 16b⟩ ≡

```
H8: r2 ∈ Amp & s2 ∈ Amp by H0,Def73;
reconsider r2,s2 as Element of Amp by H8;
H6: r2*s2 ∈ Amp by H0,GCD:def 9;
H7: t is_normalized_wrt Amp by H6,H5,H2,H1,Def73;
```

◇

Definition defined by parts 15, 16ab, 17.
Definition referenced in part 8a.

It remains to show that `t~r+s`. To do so, we only have to take into consideration that `num(r+s) = r1*s2+s1*r2` and `denom(r+s) = r2*s2` hold and to substitute `r2 = 1.G` and `s2 = 1.G` respectively to get the desired equation `num(t)*denom(r+s) = num(r+s)*denom(t)`.

⟨proof of the theorem 17⟩ ≡

```
H9: num(t)*denom(r+s) = (r1*s2+r2*s1)*denom(r+s)  by H2
                      .= (r1*s2+r2*s1)*(r2*s2)    by H0,F2
                      .= num(r+s)*(r2*s2)         by H0,F2
                      .= num(r+s)*denom(t)        by H1;
H13: t ~ (r+s) by H9,Def76;
thus thesis by H13,H7;
end;  :: M
thus thesis by M;
end;
```

◇

Definition defined by parts 15, 16ab, 17.
Definition referenced in part 8a.