MIZAR Verification of Generic Algebraic Algorithms

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
vorgelegt von
Christoph Schwarzweller

Tübingen
1997

Tag der mündlichen Qualifikation:	17.12.1997
Dekan:	Prof. Dr. Ulrich Güntzer
1. Berichterstatter:	Prof. Dr. Rüdiger Loos
2. Berichterstatter:	Prof. Dr. David Musser

Abstract

Although generic programming founds more and more attention — nowadays generic programming languages as well as generic libraries exist — there are hardly approaches for the verification of generic algorithms or generic libraries. This thesis deals with generic algorithms in the field of computer algebra. We propose the MIZAR system as a theorem prover capable of verifying generic algorithms on an appropriate abstract level. The main advantage of the MIZAR theorem prover is its special input language that enables textbook style presentation of proofs. For generic versions of Brown/Henrici addition and of Euclidan's algorithm we give complete correctness proofs written in the MIZAR language.

Moreover, we do not only prove algorithms correct in the usual sense. In addition we show how to check, using the MIZAR system, that a generic algebraic algorithm is correctly instantiated with a particular domain. Answering this question that especially arises if one wants to implement generic programming languages, in the field of computer algebra requires non trival mathematical knowledge.

To build a verification system using the MIZAR theorem prover, we also implemented a generator which almost automatically computes for a given algorithm a set of theorems that imply the correctness of this algorithm.

Abstract

Obwohl das generische Programmieren immer mehr an Bedeutung gewinnt, — heutzutage existieren generische Programmiersprachen sowie generische Bibliotheken — gibt es kaum Ansätze zur Verifikation generischer Algorithmen oder generischer Bibliotheken. In dieser Arbeit beschäftigen wir uns mit generischen Algorithmen aus dem Bereich der Computer Algebra. Wir schlagen das MIZAR Sytem als einen Beweiser vor, mit dem generische Algorithmen auf adäquater abstrakter Ebene verifiziert werden können. Der Hauptvorteil des MIZAR Systems ist seine spezielle Eingabesprache, die es erlaubt mathematische Beweise textbuchartig zu präsentieren. Für generische Versionen der Brown/Henrici Addition und des Euklid'schen Algorithmus geben wir vollständige in MIZAR formulierte Korrektheitsbeweise an.

Dabei beweisen wir nicht nur die Korrektheit des Algorithmus im üblichen Sinn. Wir zeigen ebenfalls auf, wie mit Hilfe des MIZAR Systems nachgewiesen werden kann, daß ein generischer algebraischer Algorithmus mit einer bestimmten Domaine korrekt instanziert ist. Die Beantwortung dieser Fragestellung, die insbesondere bei der Implementierung generischer Programmiersprachen auftritt, verlangt im Bereich der Computer Algebra tiefgehende mathematische Zusammenhänge.

Um das MIZAR System in ein Verifikationssystem einzubinden, haben wir außerdem einen Generator implementiert, der benutzerunterstützt aus einem gegebenen Algorithmus eine Menge von Theoremen berechnet, die die Korrektheit dieses Algorithmus implizieren.

Contents

Chapter 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++ ([Str94]); or they are even completely designed as a generic language like SuchThat ([LS96], [Sch96]). 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.

In this thesis we argue for the need of such a verification and propose the Mizar system as a theorem prover for doing so in the field of computer algebra. We give examples for its use and provide some tools to integrate Mizar into a verification system.

1.1 Generic Algorithms

Generic programming has a lot of advantages compared with non-generic programming. It supports reusability and so safes programming. In addition generic programming allows one to build up well structured libraries (see [MS96] or [Sch96]). But what exactly is meant by a generic algorithm?

In general one considers an algorithm to be generic if it is applicable with different data structures. Obviously that is a property of generic algorithms, but only a small facet of the whole. Generic programming goes much deeper, up to the essence of a method: We look for the minimal conditions and domains under which a method works — independent of any data structure.

As a rather trivial example we consider the addition of two integer polynomials. The method used here is addition of components in the coefficient domain. But that depends neither on the number of indeterminates nor on the integers themselves. It only requires the existence of addition in the coefficient domain resp. that the coefficient domain is a ring which implies the existence of addition. So, addition of polynomials is completely independent of the structure of the given polynomial ring. Consequently this should lead to one generic addition algorithm for arbitrary polynomial rings over arbitrary rings.

Especially in the field of algorithmic mathematics we can find this kind of view concerning genericity. Algorithms are written over abstract domains using only their axioms and operations (see for example [Lip71]). In fact this is generic programming: Writing algorithms for multiple of (abstract) domains having in common that they fulfill the (minimal) conditions which make the underlying method work.

1.2 The SuchThat Project

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 ([LC92]) from which it adopted its instructions. 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 a declaration mechanism that enables the user to introduce arbitrary algebraic structures. Subsequently, algorithms are written based on these structures.

We consider the Euclidean algorithm as an easy example: Euclid's method for computing the greatest common divisor relies on the observation that

$$gcd(a,b) = gcd(b, a \bmod b) \text{ for } b \neq 0 \text{ and } gcd(a, 0) = a.$$

Algorithms for greatest common divisors over the integers or polynomial rings over fields use this method by computing a remainder sequence of a and b until 0 appears and the second equation is applicable. Now, what are the abstract conditions making this method work? Of course we need a quotient-remainder-function to compute the remainder sequence, hence the domain has to be a ring. But that is not enough. What guarantees the termination of this method? Intuitively speaking the decreasing size of the remainders. This is given in an abstract way by a degree function. Consequently, to compute greatest common divisors with Euclid's method we need the underlying structure to be a — therefore so-called — Euclidean domain with its corresponding degree function:[1]

```
"eucl.sth" 5 ≡
    global: let E be EuclideanRing;
            let d be DegreeFunction of E;
            let Amp be AmpleSet of E.

    Algorithm: c := GCD(a,b)
    Input:     a,b ∈ E.
    Output:    c ∈ E such that c ∈ Amp & c = gcd(a,b).

    local: u,v,s,t ∈ E.

    (1) [Initialization]
        u := a;
        v := b.
```

[1] Note that we actually analysed the method of Euclid — to use the vocabulary of [Sch96]: we lifted the integer Euclidean algorithm — and did not develop a general generic method for computing greatest common divisors. In fact there are gcd domains in which greatest common divisors cannot be computed using Euclid's method (see [SL95]).

```
    (2) [a = 0]
        if u = 0 then {c := NF(v); return;}.
    (3) [Loop]
        while v ≠ 0
          { QR(u,v;s,t);
            u := v;
            v := t}.
    (4) [Normalization]
        c := NF(u). □
◇
```
File defined by parts 5, 6ab.

We feel that the use of the global parameter `AmpleSet of E` needs some further explanation. Its use is due to an algorithmic problem: In general there is more than one element in an Euclidean domain fulfilling the definition of the greatest common divisor. In contrast the result of an algorithm should be unique.

One possibility is to compute all greatest common divisors, the result of the algorithm then being a subset of the Euclidean domain. But this is impractical thinking of a greatest common divisor algorithm as a subalgorithm whose result shall be further processed. So, let us again look at the integers. Here the solution is to give side conditions: A greatest common divisor greater or equal than zero is computed. This goes along with the concept of ample sets. (In fact the non negative integers form an ample set for the integers.) Any two greatest common divisors are associates of each other. The association relation divides the Euclidean domain into equivalence classes and an ample set is a subset of the Euclidean domain that contains exactly one element from each equivalence class of associates.[1] Consequently, asking for a greatest common divisor that is a member of the ample set, leads to a unique result of the algorithm.

We also give the specifications of the subalgorithms. Note that the methods of these subalgorithms have no influence on the correctness of the Euclidean algorithm. Only when the algorithm is instantiated, there has to be a subalgorithm fulfilling this specification (over the current domain).

Here is the specification of the quotient remainder function:

"eucl.sth" 6a ≡
```
    Algorithm: QR(x,y;q,r)
    Input:     x,y ∈ E such that y ≠ 0.
    Output:    q,r ∈ E such that x = q*y+r & (r = 0 or d(r) < d(y)). □
◇
```
File defined by parts 5, 6ab.

To compute the normal form modulo an ample set we use — according to the paradigm of genericity — more general structures than Euclidean domains: The association relation is defined on integral domains, hence also normal forms should be defined on integral domains. Consequently, we have to introduce new global declarations before we can state the specification of the normal form subalgorithm. We use ~ to denote the association relation.

[1] In fact one can define ample sets over arbitrary sets and arbitrary equivalence relations.

```
"eucl.sth" 6b ≡
    global: let I be integral domain;
            let Amp be AmpleSet of I.

    Algorithm: y := NF(x)
    Input:     x ∈ I.
    Output:    y ∈ I such that y ∈ Amp & y ~ x. □
◇
```
File defined by parts 5, 6ab.

Note that the just given Euclidean algorithm is by no means an abstract algorithm, but a generic algorithm — written in SuchThat — which indeed can be instantiated and executed.

For execution SuchThat algorithms are translated into C++. Due to its template mechanism C++ is suitable to represent generic (algebraic) algorithms. Once compiled, SuchThat programs can be instantiated with special domains in the usual way.

The problem with C++ templates is that type parameters are not checked for correctness. So calling our example algorithm with an ordinary ring (or even with a group) lacking the necessary degree function gives an error only at runtime. To detect such kinds of errors already at compile time, the SuchThat compiler includes a type checker: Based on the given declarations it checks whether the present instance is algebraically correct. In the example we would have to determine whether the integers form an Euclidean domain (and whether there exist algorithms for computing `QF` and `NR` over the integers). To answer these questions the SuchThat compiler is equipped with an algebraic data base holding the necessary algebraic information.

To summarize, SuchThat is a programming language that enables writing abstract algebraic algorithms in the sense of [Lip71] nevertheless being executable programs.

1.3 Example: Generic Brown/Henrici Addition

In this section we consider as another example the algorithm of Brown and Henrici concerning addition of fractions over gcd domains.

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 arithmetic 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 each two elements a greatest common divisor exists. We also assume that there are algorithms `fract` to construct a fraction out of Elements of `I` and algorithms `num` and `denom` that decompose a fraction into numerator and denominator respectively.

The algorithm accepts normalized fractions as input, giving as the result again 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. This allows singling out trivial cases leading in general to more efficient runtime behaviour (see [Col74]).

"BrHenAdd.sth" 8a ≡

```
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: r1,r2,s1,s2,d,e,r2',s2',t1,t2,t1',t2' ∈ I;

(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 8ab.

Please note that in general we do not have `t = r+s`, but only `t ~ r+s`, which means `num(t)*denom(r+s) = denom(t)*num(r+s)`.[1] The reason for this is that the fraction `t = r+s` is defined as usual by `num(t) := num(r)*denom(s)+denom(r)*num(s)` and `denom(r+s) := denom(r)*denom(s)`; hence `r+s` is no normalized fraction in general and `t = r+s` cannot serve as the output specification of the algorithm.

The correctness of the algorithm depends on deep properties of greatest common divisors (see [Col74]). We will see in the following how to prove them (and correctness of the algorithm) rigorously with machine assistance.

We conclude with the specifications of the subalgorithms. Note that the Euclidean algorithm of section 1.2 satisfies the specification of the greatest common divisor function.

"BrHenAdd.sth" 8b ≡

```
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). □
```

[1] Compare the corresponding MIZAR definitions in section 4.4.

```
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 8ab.

1.4 Proving Generic Algorithms Correct

We have seen that the paradigm of genericity allows one to develop extremely powerful algebraic algorithms. On the other hand generic programming requires a more careful verification — especially if a generic algorithm will be kept in a library. From our point of view generic algorithms introduce two kinds of correctness:

First there is correctness in the usual sense; that is, an algorithm has to fulfill its specification. Besides, this proof has to be done on an abstract level: We need generic correctness proofs for generic algorithms to cover all possible instantiations of the algorithm's parameters in the proof. For example to prove the generic addition algorithm of Brown and Henrici correct, we have to argue over gcd domains, so just using the axioms of a gcd domain and nothing else.

The second kind of correctness concerns the use of generic algorithms: Is a particular instantiation correct with respect to the specification of a generic algorithm? Obviously, if a generic algorithm is called with a particular domain, the result is correct if and only if the domain fulfills the requirements of the specification. For example, if the generic Brown/Henrici algorithm of the last section is instantiated with a polynomial ring, we have to check whether this ring is a gcd domain. This seems to be a version of the type problem in typed programming languages; in fact it can be seen as a type problem (see [Sch96]). Nevertheless here we comprehend it as a matter of correctness, because the questions to be answered differ extremely from the ones in ordinary type checking: They include the use of mathematical theorems.

As another example consider a generic algorithm which computes the absolute value function over an ordered semigroup. If this algorithm is called, we have to check whether the integers or whether the integers modulo p are a semigroup, which in addition allow the required order.

We believe that both kinds of correctness are important for generic programming. Especially in the field of computer algebra the requirements of algorithms are in no way trivial. In addition these requirements concern not only domains, but also the input/output parameters themselves.

SacLib ([CL90]) for instance contains (non generic) algorithms for factoring polynomials. Algorithm IUSFPF — integral univariate square-free polynomial factorization — expects as input an integral univariate square-free polynomial, which also is positive, primitive and of positive degree. The result is a list of the positive irreducible factors of the input polynomial. Thus thinking of a generic algorithm for this task (for instance a lifted version of IUSFPF), we see that establishing correctness of (generic)

algebraic algorithms is by means a nontrivial mathematical process.

Consequently developing generic algebraic algorithms and their verification should go hand in hand.[1] In particular it is of considerable advantage if correctness is proved by the same people developing the algorithm.

A theorem prover for supporting verification of generic algebraic algorithms must take this view into account: The gap between the language of algebra (in which we write algorithms and argue about their correctness) and the language of the theorem prover (in which we formally prove correctness of our algorithms) should be as small as possible.

Unfortunately, present computer algebra systems like AXIOM ([JS92]), though able to express quite complex algebraic domains, do not include proof assistance for mathematical theorems. On the other hand powerful theorem provers like HOL ([Hol94]) require the knowledge of a proof logic and special proof tactics, topics distinct from the language of algebra.

We propose the MIZAR system ([Rud92]) as a theorem prover suitable for supporting verification of generic algebraic algorithms. MIZAR[2] is a system that — originally intended for support in writing mathematical papers — admits expressing mathematical knowledge in a very natural style. It also includes a large library of MIZAR articles and a checker that verifies articles written in the MIZAR language. Therefore the user is able to naturally formulate and prove theorems that arise when verifying generic algebraic algorithms. The main goal of this thesis is to establish MIZAR as a theorem prover in the field of computer algebra and generic algebraic algorithms.

To prove generic algebraic algorithms correct using the MIZAR system, we need to know which theorems we have to prove. Given an algorithm and its specification we are far away from the actual theorems ensuring correctness of the algorithm. We need to construct a so-called *verification condition set*: a set of theorems that imply the correctness of the original algorithm (see [Dil94]).

The classical method that allows one to compute such verification condition sets is the *calculus of Hoare* ([Hoa69]). Using this calculus one deduces triples of the form $\{P\}A\{Q\}$ with the meaning that program A is correct with respect to precondition P and postcondition Q. A Hoare calculus derivation depends on mathematical theorems in the following way: To apply some of the calculus' rules, certain theorems of the underlying theory must hold. Consequently, these theorems serve as a verification condition set for a given generic algorithm. Moreover, we can prove exactly these theorems using the MIZAR system, thus proving that for a given generic algebraic algorithm and its specification it is possible to construct a Hoare calculus derivation.

Evolving algebras ([Gur93]) are a promising tool for describing algorithms on abstract levels. They have been used to define operational semantics of programming languages as well as to specify real-time systems, compilers, architectures and much more. Evolving algebras are abstract state machines that transform a given state into

[1] We think of using literate programming ([Knu84]), so that generic algorithms, their documentation and their verification are combined in one document (see also [Sim97]). The algorithm resp. the verification part then can be extracted for further processing like compilation resp. proof checking.

[2] See also the MIZAR home page `http://mizar.uw.bialystok.pl`.

another one using term-based transition rules of the form

$$\textbf{if } t_0 \textbf{ then } f(t_1, \ldots, t_n) := t_{n+1} \textbf{ endif},$$

where t_0, $f(t_1, \ldots, t_n)$ and t_{n+1} are terms over a given signature. Evolving algebras allow so-called *external functions*: Functions that are not affected by the transition rules, but determine their values by an oracle. Consequently, one can describe generic algebraic algorithms at the appropriate level by introducing such external functions over the necessary algebraic domains.[1]

Unfortunately, so far evolving algebras are a rather theoretical tool. Although there exist many papers using this approach,[2] we only found a few interpreters for evolving algebras. Especially environments for working with evolving algebras or for proving properties about algorithms specified with evolving algebras are still under development. Consequently, we decided to implement a Hoare calculus based verification condition generator — not at least to get quickly theorems that allow showing the power of the MIZAR system in the field of generic algebraic programming.

1.5 Overview

The organization of the thesis is as follows: In chapter two we describe the MIZAR system in more detail. We show the structure of a MIZAR article and give some example proofs of algebraic theorems (which we will need to prove correctness of the generic Brown/Henrici addition algorithm).

In chapter three we present a Hoare-calculus-based verification condition generator, which is able to construct automatically verification conditions for generic Brown/Henrici addition and for the generic Euclidean algorithm provided that the loop invariant is given. It is implemented in SCHEME.

The following two chapters give examples for using MIZAR to prove generic algebraic algorithms correct. Chapter four contains the verification of the generic Brown/Henrici addition algorithm of section 1.3. For that we give MIZAR proofs for the verification conditions constructed by the generator of section three. In chapter five the generic Euclidean algorithm presented in section 1.2 is verified.

In chapter six we discuss how to use MIZAR for algebraic typechecking and give an example proof related to the generic Euclidean algorithm.

Finally, after a short summary we suggest some further work, especially some necessary tools to integrate the MIZAR theorem prover into a verification system for generic algebraic algorithms.

[1] For instance the subalgorithms `QF` and `NF` of the Euclidean algorithm from above could be modeled by external functions.

[2] See [Boe95] or `http://www.eecs.umich.edu/gasm` for an overview.

Chapter 2

The MIZAR System

MIZAR [Rud92] is a theorem prover based on natural deduction (see [Kle67]). Starting from the axioms of set theory.[1] and some axioms of the real numbers, up to now about 20,000 theorems from such different fields of mathematics as topology, algebra, category theory and many more have been proven and stored in a library.

From our point of view the main contribution of the MIZAR system is its special proof script language. This language is declarative[2] and associates the natural deduction steps with English constructs, thus allowing to write proofs close to textbook style.

A small example shall illustrate what we mean: Consider the following piece of MIZAR code:

`let` x `be` α;
`assume` $\varphi(x)$;
⟨*proof of* $\psi(x)$⟩

Actually, this is a proof of $\forall x.\alpha : \varphi(x) \Longrightarrow \psi(x)$ — or better of `for` x `being` α `holds` $\varphi(x)$ `implies` $\psi(x)$ as written in the MIZAR language. The structure of this proof exactly corresponds to the one mathematicians use: To prove an $\forall-$quantification $\forall x.\alpha : \theta(x)$, one takes a arbitrary but fixed element a of type α and proves $\theta(a)$. Furthermore, to prove an implication $\varphi \Longrightarrow \psi$, it is rather obvious to suppose that formula φ holds and then to prove ψ.[3]

Now, although natural deduction captures many mathematical idioms, it is not ideal for every application. Often we are not interested in every individual natural deduction step. For example to prove $a \cdot 0 = 0$ in an integral domain I, we do not want to write the exact detailed sequence of deduction steps, but do the proof in one step using the theorem $\forall x \in I : x \cdot 0 = 0$. That is, we want to take obvious shortcuts using knowledge that already has been proven elsewhere. This is exactly what the MIZAR proof checker does. We may write

$$\varphi \ \texttt{by}\ L_1, \ldots L_n;$$

with the meaning that φ is an obvious consequence of the theorems $L_1, \ldots L_n$. Furthermore the L_i may be labeled steps in the present deduction sequence or already

[1] To be more precise, it starts from the axioms of a variant of ZFC set theory due to Tarski (see [Tar38]).

[2] See [Har97] for a discussion of declarative and procedural proofs.

[3] In fact this is an application of the so called deduction theorem $\vdash \varphi \Rightarrow \psi$ *iff* $\varphi \vdash \psi$.

proved theorems. Thus a new MIZAR proof starts on the appropriate level (provided that the theorems on which the proof is built already exists in the MIZAR library).

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;
```

Because `domRing` is defined as a commutative ring which fulfills the integral attribute, it inherits all properties of (commutative) rings — especially all preproved theorems concerning rings are applicable to `I`. So the theorem mentioned above — of course being valid for arbitrary rings `R` — in the MIZAR language looks like this:

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

and 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, so using only arguments that hold for every possible instantiation. Hence we prove the algorithms correct on the appropriate abstract algebraic level.

So due to its natural proof script language, MIZAR is 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 go deep into a proof logic or the tactics of a special theorem prover.

In the following sections we describe MIZAR in detail, give some example proofs and continue our example algorithm of section 1.3. We use STWEB, a simple literate programming tool derived from NUWEB [Bri89], which allows the extraction of the MIZAR code from this document.

2.1 Introduction

In this section we describe the overall structure of a MIZAR article and show how naturally mathematical knowledge can be formalized in MIZAR. As examples we give extracts from the article `GCD.MIZ`, in which we prove theorems that are crucial for the correctness of the generic addition algorithm of Brown and Henrici.

Each MIZAR article consists of two main parts: the *environment* part and the *text proper*.[1]

[1] To bring more structure into a MIZAR article, it is possible to have more than one `begin` in the text

"gcd.miz" 15a ≡

```
environ
 ⟨environment 15b⟩
begin
 ⟨text proper 16, ... ⟩
```

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

The environment consists of several *directives* indicating which items of the MIZAR library can be referenced in the text proper. By means of these directives the knowledge stored in the library is made available for the present article. Each key is followed by a list of MIZAR article names.

⟨environment 15b⟩ ≡

```
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;
definitions
STRUCT_0;
theorems
TARSKI,BOOLE,WELLORD2,SUBSET_1,ENUMSET1,VECTSP_1,VECTSP_2;
clusters
STRUCT_0,VECTSP_1,VECTSP_2;
schemes
SETFAM_1,GROUP_2;
```

◇

Definition referenced in part 15a.

The directive `vocabulary` adds symbols of the named files to the article's internal lexicon. If there are new symbols (introduced in text proper) these have to be put in an extra vocabulary file like `GCD.VOC` in this case.[2]

The directives `notations` and `constructors` request the conceptual framework of the article. In MIZAR it is possible to introduce synonyms, if another name is more appropriate in the current context. So the `constructors` directive gives the concepts to be used, and the `notations` directive gives the synonyms to be used for these concepts. The `clusters` directive will be explained in section 4.3.

The `definitions` and `theorems` directives indicate which definitions and theorems may be cited in the article to justify reasonings. The directive `schemes` describes second order theorems that can be referenced in the text proper.

The text proper includes the new mathematical knowledge; that is, new definitions and theorems as well as proofs for these. *Reservations* declare the (mathematical) type of identifiers from the point of reservation up to the end of the article or until the reservation is overwritten by a new one:

proper. Then each `begin` stands for a new section of the article. But note that this is not necessary and that `begin`s can be introduced at places we would not consider as the starting point of a new section.

[2] See the beginning of appendix A for a description of this file.

⟨text proper 16⟩ ≡

```
reserve X,Y,Z for set;
reserve I for domRing;
reserve a,b,c,d for Element of the carrier of I;
```

◇
Definition defined by parts 16, 123.
Definition referenced in part 15a.

After this reservation `I` stands for an integral domain and `a`,`b` and `c` are Elements of `I`. As a consequence every theorem about integral domains that already has been proved in an arbitrary MIZAR article (that is referenced in the environment) can be applied to them. Using these identifiers new concepts of integral domains can be defined, for instance divisibility:

"gcd.miz" 17a ≡

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

◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

Note that `divides` is a new vocabulary item that as mentioned above must be introduced in a vocabulary file. Of course such definitions can be used to define further predicates (or special functions):

"gcd.miz" 17b ≡

```
definition
let I be domRing;
let x be Element of the carrier of I;
pred x is_unit means :Def2:
 x divides 1.I;  :: 1.I is the multiplicative identity of I
end;

definition
let I be domRing;
let a,b be Element of the carrier of I;
pred a is_associated_to b means :Def3:
 a divides b & b divides a;
antonym a is_not_associated_to b;
end;
```

◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

The phrase `antonym` allows us to introduce a synonym for the negation of the just defined predicate. Also, one can define functions over algebraic domains, but contrary to defining predicates one has to prove existence and uniqueness of this new defined function:[1]

[1] Note that the definition not only states properties about the resulting value, but also its type.

"gcd.miz" 18a ≡
```
    definition
    let I be domRing;
    let x,y be Element of the carrier of I;
    assume d: y divides x & y <> 0.I;
      func x/y -> Element of the carrier of I means :Def5:
      it*y = x;  :: it stands for the value of the defined function.
    existence
      proof
      H1: ex z being Element of the carrier of I
          st x = y*z by d,Def1;
      thus thesis by H1;
      end;
    uniqueness
      by d,IDOM1;
      :: theorem IDOM1 states that I fulfills the cancellation property.
    end;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

One of the main advantages of the MIZAR language is that it allows us to formulate theorems (and, as we will see later, proofs) in mathematical textbook style using the just defined concepts, hence giving the possibility to work with a proof checker using the language of algebra:

"gcd.miz" 18b ≡
```
    theorem
    L1:for I being domRing
    for a,b,c being Element of the carrier of I holds
    a divides a &
    ((a divides b & b divides c) implies a divides c)
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

`L1` is a label that can be used to refer to this theorem in a justification.[1] So once proved for arbitrary integral domains `I`, this theorem is applicable to every object whose type widens to `domRing`; that is to every object fulfilling the properties of mode `domRing`. This includes for example mode `gcdDomain` and mode `EuclideanRing` because they are defined as `domRing` with additional properties.

In the next section we will see, how such a theorem is proved in MIZAR.

2.2 Proving Algebraic Theorems

In this section we want to show in detail how theorems are proved in MIZAR. To be more precise, we want to illustrate that such proofs are closer to textbook proofs than proofs of other mechanized systems in terms of the syntax in which they are stated. Hence using the MIZAR language one can formulate proofs directly in the language of algebra.

As a first example we prove theorem `L1` of the last section. We start by introducing the domains and the elements the theorem is about:

"gcd.miz" 19a ≡
```
    proof
    let I be domRing;
    let a,b,c be Element of the carrier of I;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

[1] Compare page 11.

Theorem L1 consists of two statements, the second of them being an implication; so we suppose that its assumption holds and prove that the conclusion follows.

"gcd.miz" 19b ≡
```
   M1: now assume
   H1: a divides b & b divides c;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

The next step is to expand the definition of divides into an existence statement. In MIZAR existence elimination is done using consider. Note that the existence of the element introduced by consider has to be justified, by the definition of divides in this case.

"gcd.miz" 19c ≡
```
   consider d being Element of the carrier of I such that
   H2: a*d = b by H1,Def1;
   consider e being Element of the carrier of I such that
   H3: b*e = c by H1,Def1;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

Using the just introduced elements and arithmetics of integral domains,[1] we get our assertion by applying once again the definition of divides:

"gcd.miz" 20a ≡
```
   H4: a*(d*e) = (a*d)*e  by VECTSP_1:def 16
               .= b*e     by H2
               .= c       by H3;
   thus (a divides b & b divides c) implies a divides c by H4,Def1;
   end;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

The first statement of the theorem is an immediate consequence of our definitions and the properties of an integral domain, and we conclude the proof with:

"gcd.miz" 20b ≡
```
   M2: a*1.I = a by VECTSP_2:1;
   M3: a divides a by M2,Def1;
   thus thesis by M1,M3;
   end;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

We see that this proof exactly follows the argumentation used by mathematicians. Consequently scientists writing generic algebraic algorithms can develop correctness proofs in MIZAR right alongside the proofs they have in mind. As another example, we give a MIZAR proof of the following well known theorem.

[1] The theorem we use has not been proven for integral domains, but for arbitrary associative group structures. It is due to the MIZAR type system that we can use it here.

⟨example lemma 20c⟩ ≡

```
theorem
L11:for a,b being Element of the carrier of I holds
a is_associated_to b iff (ex c st (c is_unit & a*c = b))
⟨proof of theorem L11 20d⟩
```

◇

Definition referenced in part 124.

In the MIZAR proof script language an *if and only if* statement has to be proven by two implications, so the overall structure of the above theorem's proof is simply as follows.

⟨proof of theorem L11 20d⟩ ≡

```
proof
K1: for a,b being Element of the carrier of I holds
    a is_associated_to b implies
    (ex c being Element of the carrier of I st (c is_unit & a*c = b))
    ⟨proof of theorem L11, if 21b⟩
K2: for a,b being Element of the carrier of I holds
    (ex c being Element of the carrier of I st (c is_unit & a*c = b))
    implies a is_associated_to b
    ⟨proof of theorem L11, only if 21a⟩
thus thesis by K1,K2;
end;
```

◇

Definition referenced in part 20c.

The proof of the *only if* direction (K2) is straightforward: It only requires application of definitions and some arithmetic.

⟨proof of theorem L11, only if 21a⟩ ≡

```
proof
let a,b be Element of the carrier of I;
assume H1: (ex c st (c is_unit & a*c = b));
consider c being Element of the carrier of I such that
H2: c is_unit & a*c = b by H1;
H3: c divides 1.I by H2,Def2;
consider d being Element of the carrier of I such that
H5: c*d = 1.I by H3,Def1;
H6: a = a*1.I   by VECTSP_2:1
     .= a*(c*d) by H5
     .= (a*c)*d by VECTSP_1:def 16
     .= b*d     by H2;
H7: b divides a by H6,Def1;
H8: a divides b by H2,Def1;
thus thesis by H7,H8,Def3;
end;
```

◇

Definition referenced in part 20d.

The proof of the *if* direction (K1) starts as usual with introducing the required elements and the application of the corresponding definitions, in order to show by using properties of the integral domain that the assertion holds:

⟨proof of theorem L11, if 21b⟩ ≡

```
proof
let a,b be Element of the carrier of I;
assume H0: a is_associated_to b;
H2: a divides b & b divides a by H0,Def3;
consider c being Element of the carrier of I such that
H5: b = a*c by H2,Def1;
consider d being Element of the carrier of I such that
H6: a = b*d by H2,Def1;
⟨cases of theorem L11, if 22a⟩
```

◇

Definition referenced in part 20d.

But then — as indicated by the name of the macro — we have to distinguish whether `a = 0.I` or whether `a` $\neq$ `0.I`. To do so the MIZAR language has a special feature: the `cases` phrase.

⟨cases of theorem L11, if 22a⟩ ≡

```
M: now per cases;
case A: a <> 0.I;
⟨proof of theorem L11, if, case A 22b⟩
case B: a = 0.I;
⟨proof of theorem L11, if, case B 23a⟩
end;  ::cases
thus thesis by M;
end;
```

◇

Definition referenced in part 21b.

Here it is obvious for the MIZAR proof checker that `A` and `B` together cover all possible cases. But it may happen that this must be proved before and referenced at level `M`. Now the rest of the proof is easy: In both cases it only requires some arithmetic followed by an application of the definition of `unit`.

Here is the proof of the case `a` $\neq$ `0.I`: Combining the introduced elements `c` and `d` gives `c * d = 1.I` by cancelling `a`, hence that `c` is a unit.

⟨proof of theorem L11, if, case A 22b⟩ ≡

```
H7: a = b*d      by H6
     .= (a*c)*d  by H5
     .= a*(c*d)  by VECTSP_1:def 16;
H8: c*d = 1.I by H7,L10,A;
H9: c divides 1.I by H8,Def1;
H10: c is_unit by H9,Def2;
thus thesis by H10,H5;
```

◇

Definition referenced in part 22a.

What follows now is the proof of the other case `a = 0.I`: We show that `b = a * 1.I`, thus getting the thesis because `1.I` is a unit.

⟨proof of theorem L11, if, case B 23a⟩ ≡

```
H1: b = a*c    by H5
      .= 0.I    by B,VECTSP_2:26
      .= 0.I*1.I  by VECTSP_2:1
      .= a*1.I    by B;
H2: 1.I is_unit
    proof
    M1: 1.I*1.I = 1.I by VECTSP_2:1;
    M2: 1.I divides 1.I by M1,Def1;
    thus thesis by M2,Def2;
    end;
thus thesis by H2,H1;
```

◇

Definition referenced in part 22a.

This completes the proof of theorem `L11`. Note that the proof just given — like the one for theorem `L1` — although easy to read for human beings, is accepted by the MIZAR proof checker.

2.3 A Theorem of Brown and Henrici

In this section we give a MIZAR proof for a theorem of Brown and Henrici, which states a nontrivial property of the greatest common divisor in arbitrary gcd domains I. In MIZAR it is formulated like this:

⟨Brown/Henrici theorem 23b⟩ ≡

```
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 of Brown/Henrici theorem 27b, ... ⟩
```

◇

Definition referenced in part 146.

We present the proof for two reasons. First we want to give an extended MIZAR proof of a nontrivial algebraic theorem. On the other hand this theorem is the heart of the correctness of the above presented generic Brown/Henrici addition algorithm.

Before we can give the entire proof, we have to do some preparations.[1] We start with the MIZAR definition of the greatest common divisor function.[2] The MIZAR definition of gcd domains is included in section 4.3.

[1] The MIZAR definition of ample sets and gcd domains can be found in section 4.2 and 4.3 respectively. A couple of theorems about divisibility in integral domains and greatest common divisors in gcd domains are stated and proved in appendix A.1 and A.3 respectively.

[2] The corresponding correctness — that is existence and uniqueness — proof can be found at the beginning of appendix A.3.

⟨Definition of gcd function 24⟩ ≡

```
definition
let I be gcdDomain;
let Amp be AmpleSet of I;
let x,y be Element of the carrier of I;
  func gcd(x,y,Amp) -> Element of the carrier of I means :Def4:
         it ∈ Amp &
         it divides x &
         it divides y &
        (for z being Element of the carrier of I
         st (z divides x & z divides y) holds (z divides it));
```

⟨correctness proof of gcd function 145b⟩

◇

Definition referenced in part 146.

Note that — contrary to the greatest common divisor function in SuchThat, where it was a global parameter — this definition of the greatest common divisor function contains the ample set as one of its arguments. Considered as a function over a gcd domain the ample set obviously has to be an explicit parameter: Without choosing a fixed ample set, the value of the greatest common divisor is not uniquely determined; for example the greatest common divisor of 4 and 6 over the integers can be 2 or -2. In Mizar however, the implicit use of function parameters is restricted only to operations being part of the underlying structure like gcd domain in this case. Nevertheless, in a programming language like SuchThat one does not want to enumerate all these parameters explicitly, but rather to introduce them in terms of global declarations.

This may be compared with the algorithm's dependence on the size of memory: The result of the algorithm strongly depends on this size — giving an overflow error or the desired result. However, no one would agree to consider the size of memory as an explicit argument of an algorithm.

Consequently we are forced to make this kind of global SuchThat declarations explicit when translating SuchThat algorithms into the Mizar language.[1]

After defining gcd domains and their corresponding greatest common divisor function the next step is to establish five properties of this function originally stated in [Col74]. Using these properties we will be able to give the proof of the Brown/Henrici theorem. Note that the following properties hold for arbitrary ample sets in arbitrary integral domains I.

⟨gcd theorems 25⟩ ≡

```
theorem
T0:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
gcd(gcd(a,b,Amp),c,Amp) = gcd(a,gcd(b,c,Amp),Amp)
```

⟨proof of theorem T0 151a⟩

[1] It is possible in Mizar to define an algebraic structure `gcdDomain-with-AmpleSet`, extending the structure of gcd domains by a corresponding ample set. Over this structure the greatest common divisor function again is the usual two argumented function. But this does not go along with our view of algebraic structures, in which a gcd domain is an integral domain fulfilling the greatest common divisor property and nothing more. An ample set — especially one for a specific relation like association — is a matter of computing in such domains, and therefore should not be contained in the original structure definition of a gcd domain.

```
theorem
T1:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
gcd(a*c,b*c,Amp) is_associated_to c*gcd(a,b,Amp)
```
⟨proof of theorem T1 151b⟩

```
theorem
T2:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
gcd(a,b,Amp) = 1.I implies gcd(a,b*c,Amp) = gcd(a,c,Amp)
```
⟨proof of theorem T2 153⟩

```
theorem
T3:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
(c = gcd(a,b,Amp) & c <> 0.I) implies gcd(a/c,b/c,Amp) = 1.I
```
⟨proof of theorem T3 26a, ... ⟩

```
theorem
T4:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
gcd(a+b*c,c,Amp) = gcd(a,c,Amp)
```
⟨proof of theorem T4 154⟩

◇

Definition referenced in part 146.

Here we only give the proof of theorem `T3` as an example. We note that the other four theorems are proved in a similar way; the corresponding proofs can be found at the end of appendix A.3. The proof starts as usual with the introduction of objects the theorem is about followed by stating the given assumptions.

⟨proof of theorem T3 26a⟩ ≡

```
proof
let Amp be AmpleSet of I;
let a,b,c be Element of the carrier of I;
assume H0: c = gcd(a,b,Amp) & c <> 0.I;
```

◇

Definition defined by parts 26abc, 27a.
Definition referenced in part 25.

Our first goal is to show that `1.I` and `gcd(a/c,b/c,Amp)` are associates of each other. Therefore we introduce elements `a1` and `b1` representing the quotients `a/c` and `b/c` respectively, concluding that `c` equals `gcd(a1*c,b1*c,Amp)`.

⟨proof of theorem T3 26b⟩ ≡

```
consider a1 being Element of the carrier of I such that H1: a1 = a/c;
consider b1 being Element of the carrier of I such that H2: b1 = b/c;
M1: c divides a & c divides b by Def4,H0;
H3: a1*c = a & b1*c = b by H1,H2,Def5,M1,H0;
H5: c = gcd(a,b,Amp)       by H0
     .= gcd(a1*c,b1*c,Amp) by H3;
```

◇

Definition defined by parts 26abc, 27a.
Definition referenced in part 25.

Now we apply theorem T1, which allows us to factor c out of gcd(a1*c,b1*c,Amp). After that, cancelling c gives the desired result.

⟨proof of theorem T3 26c⟩ ≡

```
H6: gcd(a1*c,b1*c,Amp) is_associated_to c*gcd(a1,b1,Amp) by T1;
H7: c is_associated_to c*gcd(a1,b1,Amp) by H5,H6;
M3: c*1.I is_associated_to c*gcd(a1,b1,Amp) by H7,VECTSP_2:1;
H8: 1.I is_associated_to gcd(a1,b1,Amp) by M3,L15,H0;
```

◇
Definition defined by parts 26abc, 27a.
Definition referenced in part 25.

The last step consists of showing that 1.I and gcd(a1,b1,Amp) are not only associates of each other, but in fact are equal. This is done using the fact that two elements of an ample set being associates of each other must be identical.[1]

⟨proof of theorem T3 27a⟩ ≡

```
H9: gcd(a1,b1,Amp) is_associated_to 1.I by H8,L2;
H10: gcd(a1,b1,Amp) is Element of Amp by Def4;
H11: 1.I is Element of Amp by Def8;
H12: gcd(a1,b1,Amp) = 1.I by H9,H10,H11,AMP;
thus thesis by H1,H2,H12;
end;
```

◇
Definition defined by parts 26abc, 27a.
Definition referenced in part 25.

Based on the above five theorems about the greatest common divisor function, one succeeds in giving a MIZAR proof for the theorem of Brown and Henrici as follows. We start by introducing synonyms for gcd(r2,s2,Amp), r2/gcd(r2,s2,Amp) and s2/gcd(r2,s2,Amp):

⟨proof of Brown/Henrici theorem 27b⟩ ≡

```
proof
let Amp be AmpleSet of I;
let r1,r2,s1,s2 be Element of the carrier of I;
assume H1: gcd(r1,r2,Amp) = 1.I & gcd(s1,s2,Amp) = 1.I &
           r2 <> 0.I & s2 <> 0.I;
consider d being Element of the carrier of I such that
H2: d = gcd(r2,s2,Amp);
H2a: d divides s2 & d divides r2 by H2,Def4;
K: d <> 0.I by H2,H1,L12;
consider r being Element of the carrier of I such that H4: r = r2/d;
consider s being Element of the carrier of I such that H5: s = s2/d;
```

◇
Definition defined by parts 27b, 28abc, 29a.
Definition referenced in part 23b.

The proof of the Brown/Henrici theorem takes advantage of the fact that

$$\begin{aligned} &\mathtt{r2}(\mathtt{s2}/\mathtt{gcd}(\mathtt{r2},\mathtt{s2},\mathtt{Amp}) \\ &= \mathtt{gcd}(\mathtt{r2},\mathtt{s2},\mathtt{Amp})(\mathtt{r2}/\mathtt{gcd}(\mathtt{r2},\mathtt{s2},\mathtt{Amp}))(\mathtt{s2}/\mathtt{gcd}(\mathtt{r2},\mathtt{s2},\mathtt{Amp})). \end{aligned}$$

[1] See section 4.2 for a MIZAR definition of ample sets and their corresponding properties.

Consequently two applications of theorem T2 will eliminate s2/gcd(r2,s2,Amp) and r2/gcd(r2,s2,Amp) leaving the desired gcd(r2,s2,Amp) resp. d as the second argument of the gcd-function.
So, all we have to show is that the requirements for theorem T2 hold, namely that gcd(r1*s+s1*r,s,Amp) = 1.I and gcd(r1*s+s1*r,r,Amp) = 1.I.

To prove the first requirement, we use theorems T3 and T2 to conclude that

⟨proof of Brown/Henrici theorem 28a⟩ ≡

```
H9: gcd(r,s,Amp) = 1.I by H4,H5,H2,K,T3;
H7: gcd(s,s1,Amp) = 1.I
    ⟨proof of H7 30a⟩
H8: gcd(s,s1*r,Amp) = gcd(s,r,Amp) by H7,T2;
```

◇
Definition defined by parts 27b, 28abc, 29a.
Definition referenced in part 23b.

These two equations (H8 and H9) enable us to show that the above mentioned requirement holds, hence to execute the first elimination step. In the MIZAR language this looks as follows.

⟨proof of Brown/Henrici theorem 28b⟩ ≡

```
:: Requirement for theorem T2
H10:   gcd(r1*s+s1*r,s,Amp)
     = gcd(s1*r,s,Amp)       by T4
    .= gcd(s,s1*r,Amp)       by L13
    .= gcd(s,r,Amp)          by H8
    .= gcd(r,s,Amp)          by L13
    .= 1.I                   by H9;
H11: r2*s = s*(d*r)
     ⟨proof of H11 29b⟩
:: Elimination of s = s2/gcd(r2,s2,Amp)
H12:   gcd(r1*s+s1*r,r2*s,Amp)
     = gcd(r1*s+s1*r,s*(d*r),Amp) by H11
    .= gcd(r1*s+s1*r,d*r,Amp)     by H10,T2;
```

◇
Definition defined by parts 27b, 28abc, 29a.
Definition referenced in part 23b.

To prove the second requirement we proceed the same way: First we use theorems T3 and T2 to show the necessary preliminaries (H15 and H16):

⟨proof of Brown/Henrici theorem 28c⟩ ≡

```
H14: gcd(r,r1,Amp) = 1.I
     ⟨proof of H14 30b⟩
H15: gcd(r,r1*s,Amp) = gcd(r,s,Amp) by H14,T2;
H16: gcd(r,s,Amp) = 1.I by H4,H5,H2,K,T3;
```

◇
Definition defined by parts 27b, 28abc, 29a.
Definition referenced in part 23b.

Just like before we can prove that the necessary requirement for theorem T2 holds. So, we also eliminate r2/gcd(r2,s2,Amp) and complete the proof with

⟨proof of Brown/Henrici theorem 29a⟩ ≡

```
    :: Requirement for theorem T2
    H17:   gcd(r1*s+s1*r,r,Amp)
         = gcd(r1*s,r,Amp)      by T4
        .= gcd(r,r1*s,Amp)      by L13
        .= gcd(r,s,Amp)         by H15
        .= 1.I                  by H16;
    :: Elimination of r = r2/gcd(r2,s2,Amp)
    H18:   gcd(r1*s+s1*r,d*r,Amp)
         = gcd(r1*s+s1*r,d,Amp) by H17,T2;
    H19:   gcd(r1*s+s1*r,r2*s,Amp)
         = gcd(r1*s+s1*r,d,Amp)  :: Remember that d = gcd(r2,s2,Amp).
         by H12,H18;
    thus thesis by H19,H4,H5,H2;
    end;
```

◇

Definition defined by parts 27b, 28abc, 29a.
Definition referenced in part 23b.

Note again that the proof just presented is not only a thorough argumentation to show that the theorem of Brown and Henrici holds, but also an accepted proof script for the MIZAR proof checker.

We conclude this section by giving the subproofs we left out above, when proving the Brown/Henrici theorem. They are easy done by equational reasoning and using basic properties of the greatest common divisor function.

⟨proof of H11 29b⟩ ≡

```
    proof
    H0: d divides d by L1;
    H0a: d divides d*r2 by L6;
    H1: r2*s = ((1.I)*r2)*s  by VECTSP_2:1
             .= ((d/d)*r2)*s  by K,L7
             .= ((d*r2)/d)*s  by K,H0,H0a,L8
             .= (d*(r2/d))*s  by K,H2a,H0a,L8
             .= (d*r)*s       by H4
             .= s*(d*r);
    thus thesis by H1;
    end;
```

◇

Definition referenced in part 28b.

⟨proof of H7 30a⟩ ≡

```
    proof
    M1: gcd(s,s1,Amp) divides s1 by Def4;
    M2: gcd(s,s1,Amp) divides s by Def4;
    consider e being Element of the carrier of I such that
    M3: gcd(s,s1,Amp)*e = s by M2,Def1;
    M4:   gcd(s,s1,Amp)*(e*d)
       = (gcd(s,s1,Amp)*e)*d  by VECTSP_1:def 16
      .= s*d                  by M3
      .= s2                   by K,H2a,H5,Def5;
    M5: gcd(s,s1,Amp) divides s2 by M4,Def1;
    M6: gcd(s,s1,Amp) divides gcd(s1,s2,Amp) by M1,M5,Def4;
    M7: gcd(s,s1,Amp) divides 1.I by M6,H1;
    M8: (1.I)*gcd(s,s1,Amp) = gcd(s,s1,Amp) by VECTSP_2:1;
    M9: 1.I divides gcd(s,s1,Amp) by M8,Def1;
```

```
    M10: gcd(s,s1,Amp) is_associated_to 1.I by M7,M9,Def3;
    M11: gcd(s,s1,Amp) is Element of Amp by Def4;
    M12: 1.I is Element of Amp by Def8;
    thus thesis by M10,M11,M12,AMP;
    end;
```

◇

Definition referenced in part 28a.

⟨proof of H14 30b⟩ ≡

```
    proof
    M1: gcd(r,r1,Amp) divides r1 by Def4;
    M2: gcd(r,r1,Amp) divides r by Def4;
    consider e being Element of the carrier of I such that
    M3: gcd(r,r1,Amp)*e = r by M2,Def1;
    M4:   gcd(r,r1,Amp)*(e*d)
       = (gcd(r,r1,Amp)*e)*d  by VECTSP_1:def 16
      .= r*d                  by M3
      .= r2                   by K,H2a,H4,Def5;
    M5: gcd(r,r1,Amp) divides r2 by M4,Def1;
    M6: gcd(r,r1,Amp) divides gcd(r1,r2,Amp) by M1,M5,Def4;
    M7: gcd(r,r1,Amp) divides 1.I by M6,H1;
    M8: (1.I)*gcd(r,r1,Amp) = gcd(r,r1,Amp) by VECTSP_2:1;
    M9: 1.I divides gcd(r,r1,Amp) by M8,Def1;
    M10: gcd(r,r1,Amp) is_associated_to 1.I by M7,M9,Def3;
    M11: gcd(r,r1,Amp) is Element of Amp by Def4;
    M12: 1.I is Element of Amp by Def8;
    thus thesis by M10,M11,M12,AMP;
    end;
```

◇

Definition referenced in part 28c.

Chapter 3

A Verification Condition Generator

To use the MIZAR system as a tool for the verification of generic algebraic algorithms one has to construct a set of theorems that ensure the correctness of the algorithm — a set of verification conditions. These theorems then can be proved using MIZAR to verify the correctness of the given algorithm.

In this chapter we describe a *verification condition generator*; that is, a program which almost automatically constructs such theorems out of a given algorithm and its specification.

The generator is based on the Hoare calculus.[1] We start with an algorithm and its input/output specification considering this as a Hoare triple to be proved.

Showing that a triple holds requires a derivation starting with the axioms of the calculus and the given triple as its final formula. But instead of trying to find such a derivation, it is much easier to start with the entire triple: One uses the rules of the calculus in a backward manner to reduce the assertion to simpler Hoare triples, until axioms are reached. Our generator uses such so-called backward rules.

Note that we do not process the SUCHTHAT algorithm itself but its representation in an abstract syntax or, to be more precise, a parse tree of the original SUCHTHAT algorithm represented in SCHEME. There are two reasons for this:

The SUCHTHAT compiler uses SCHEME as an intermediate language into which each SUCHTHAT algorithm is translated. So starting with this representation frees us from parsing, performing syntax checks and so on. All these things are done by the SUCHTHAT compiler. Nevertheless we included some of these checks in order to use the generator in a stand-alone mode, too.

On the other hand using abstract syntax makes the generator programming language independent in the sense that the generator is applicable to every programming language that can be translated into this representation. And it is a minor task to construct a SCHEME representation out of a parse tree independently of the original programming language.

[1] For an introduction to the Hoare calculus see [Dil94] or [Hoa69].

The generator is divides its task into three parts:

- annotating the algorithm
- constructing abstract theorems
- constructing specific theorems

The first step consists of annotating the given algorithm: we introduce abstract intermediate predicates P_i in sequences and invariants P_j for loops. This allows to apply Hoare's rules in a backward manner.

Subsequently theorems are constructed: First abstract theorems are generated based on the Hoare calculus. By abstract theorems we mean theorems with variables P_i for the predicates. As a consequence up to this point we can ensure the correctness of the theorems; that is, these theorems imply the correctness of the given algorithm due to Hoare calculus.

The last thing to do is to find specific counterparts for the abstract predicates with the input/output specification as a starting point. We have implemented some heuristics to get simple theorems, and there are two points worth mentioning:

- Not every abstract predicate will be specialized (e.g. loop invariants lie in the responsibility of the user). Before starting this part of the generator the user has the possibility to set predicates by hand. The generator then tries to fill in the remaining predicates with respect to the user given ones.
- The results of the generator should be considered as an aid to the user. Contrary to the abstract theorems there is not always a guarantee that the specialized theorems hold, because the chosen specific predicate may be not suitable (but if the specialized theorems hold they ensure the correctness of the given algorithm).

In the following sections we describe the structure of the verification condition generator. Note that we use StWeb so that the Scheme code can be extracted from this document.

3.1 The Kernel of the Generator

The construction principle behind the generator is that of viewing the given algorithm as an object on which different activities are performed. These activities may generate theorems for the algorithm or may do something completely different. So to annotate a given algorithm there is nothing to be done but calling the corresponding activities.

"kernel.scm" 35a ≡

```
(define (annotate prog)
   (do-activities prog 'annotations))
```

◇

File defined by parts 35ab, 39b, 40ab.

We proceed the same way, whether we want to construct abstract theorems or get conjectures for the abstract theorems:

"kernel.scm" 35b ≡
```
    (define (generate-theorems annotated-prog)
       (do-activities annotated-prog 'generations))

    (define (guess annotated-prog)
       (do-activities annotated-prog 'guesses))
```
◇
File defined by parts 35ab, 39b, 40ab.

The activities that have to be performed depend on the kind of the given program statement (for example whether it is a *while*-construct or a simple *assignment*). We use the `alist` package of the SLIB[1] ([ELJ94]) to store this information, namely information about the format of the construct, of how to annotate it, of how to generate abstract theorems and of how to find conjectures for abstract theorems.

"initialize-tables.scm" 35c ≡
```
    (require 'alist)

    (define formats '())
    (define annotations '())
    (define generations '())
    (define guesses '())
```
◇
File defined by parts 35c, 184ab, 185ab, 186b.

How are activities for annotating algorithms and generating theorems represented? An activity is a SCHEME list consisting of an *activity name* and an optional number of further arguments describing the activity in more detail.

As an example we consider the *while*-construct `(while condition action)`: To annotate this, we have to introduce an intermediate predicate after `condition` (the loop invariant) and to recursively annotate `action`.[2] This is written (in SCHEME) as follows.

⟨while annotations 36a⟩ ≡
```
    '((rec 'action)
      (insert-pred-after 'condition))
```
◇
Definition referenced in part 36c.

Constructing theorems for the *while*-construct is based on the *while rule* of the Hoare calculus:

$$\frac{\{inv \wedge b\}\, S\, \{inv\},}{\{inv\}\mathbf{while}\ b\,\mathbf{do}\ S; \{inv \wedge \neg b\}}$$

When computing verification conditions in a backward manner, in general we will not have inv nor b itself. In fact we will have nothing more but arbitrary predicates P and Q. As a consequence, we will have to prove $\{P\}$ **while** $b\ S\,\{Q\}$, using in addition the so-called *implication rule*:

$$\frac{P \to P',\ \{P'\}\ S\ \{Q'\},\ Q' \to Q}{\{P\}\ S\ \{Q\}}$$

[1] We used GAMBIT SCHEME Version 2.5.1 and SLIB version 2a2.
[2] Compare chapter 3.3.

These two rules together show that to verify an arbitrary *while*-statement, it suffices to show $P \rightarrow inv$, $\{inv \wedge b\}\, S\, \{inv\}$ and $(inv \wedge \neg b) \rightarrow Q$.[1] In our representation we get the following list of activities:

⟨while generations 36b⟩ ≡

```
'((theorem-is 'pre 'inv)
  (theorem-is '(and inv (not condition)) 'post)
  (rec '((and inv condition) action inv)))
```

◇

Definition referenced in part 36c.

New activities can be added with `insert-newconstruct`. Besides the activities one has to specify a key and the format of the construct.

⟨insert rules 36c⟩ ≡

```
(insert-newconstruct 'while
     '(while condition action)
     ⟨while annotations 36a⟩
     ⟨while generations 36b⟩)
```

◇

Definition defined by parts 36c, 38b, 39a, 41a.
Definition referenced in part 185a.

Note that to expand the generator with a new programming language construct (using `insert-newconstruct`) one only has to add the rules for processing it.

We shall take the *procedure call* as a second example, also to describe the rule we implemented to prove such a call correct. For that we need the notion of *substitution*: To substitute a variable x by a term t in a formula (or another term) s, one replaces each free occurrence of x in s by t. We denote the resulting formula (term) by $s_x[t]$. We require a procedure to have no side effects; that is, the only variables changed by a procedure call are the ones explicitly given in the output specification. We further assume that

$$\{input\text{–}specification(f)\}\, f(\vec{x}; \vec{z})\, \{output\text{–}specification(f)\}$$

where $\vec{x}$ stands for the formal input parameters and $\vec{z}$ for the formal output parameters of procedure f, is a valid Hoare formula.[2] To prove a specific procedure call correct, we adopted the following theorem for procedure calls from [Gri81]:

Let I be a predicate and let procedure f be correct with respect to its specification as mentioned above. Assume that none of the free identifiers in I appear in the output variables $\vec{z}$ of f. Then holds

$$\{input\text{–}specification(f)_{\vec{x}}[\vec{a}] \wedge I\}\, f(\vec{a}; \vec{c})\, \{output\text{–}specification(f)_{\vec{x},\vec{z}}[\vec{a}, \vec{c}] \wedge I\}.\ \square$$

The predicate I captures the notion of *invariance*: predicates that do not refer to the output variables of a subalgorithm remain unchanged throughout the procedure call. Requiring I to be invariant for procedure f of course restricts the procedure call. On the other hand this rule is easier to handle and allows for better constructing of

[1] Some authors refer to this as the *derived while rule*.
[2] Thus, we suppose subalgorithms to be correct with respect to their specification.

specific predicates. Furthermore the restriction will be trivially fulfilled if the output variables of the procedure call are fresh, which is the case in most procedure calls in algebraic algorithms (see our examples).

To prove that $\{P\}\ f(\vec{a};\vec{c})\ \{Q\}$ resp. $\{P\}\ c := f(\vec{a})\ \{Q\}$ holds for arbitrary predicates P and Q,[1] we proceed as we did with the *while rule*: Again we integrate the *implication rule* into the activities, getting the following three conditions to prove.

- $P \longrightarrow \mathit{input\text{–}specification}(f)_{\vec{x}}[\vec{a}]$,
- $(P \wedge \mathit{output\text{–}specification}(f)_{\vec{x},\vec{z}}[\vec{a},\vec{c}]) \longrightarrow Q$ resp.
  $(P \wedge \mathit{output\text{–}specification}(f)_{\vec{x},z}[\vec{a},c]) \longrightarrow Q$ and
- None of the free identifiers of P equals z resp. appear in $\vec{z}$.

The straightforward translation of these conditions into our SCHEME representation gives the following activities for procedure call.

⟨procedure call generations 38a⟩ ≡

```
'((is-invariant-for 'pre '(outputparam proc))
  (theorem-is 'pre
              '(subst (inputspec 'proc)
                      (formalparam 'proc)
                      (actualparam 'proc)))
  (theorem-is '(and pre
                    (subst (outputspec 'proc)
                           (formalparam 'proc)
                           (actualparam 'proc)))
              'post) ))
```

◇

Definition referenced in part 38b.

Because there is nothing to do to annotate a procedure call, inserting the procedure call activities looks as follows. (The star in the format definition indicates that any number of arguments will be accepted.)

⟨insert rules 38b⟩ ≡

```
(insert-newconstruct 'call
     '(call proc * *)
     'none
     ⟨procedure call generations 38a⟩
```

◇

Definition defined by parts 36c, 38b, 39a, 41a.
Definition referenced in part 185a.

Yet to be described are activities for the *assignment*, *if*, *sequences*, and *return* constructs. Note that the activities for generating theorems directly mirror the corresponding Hoare rules.[2]

[1] Note that $z := f(\vec{x})$ is just another syntax for $f(\vec{x};z)$.

[2] The *return rule* states that the predicate holding before the `return` is executed implies the post-condition of the original algorithm.

⟨insert rules 39a⟩ ≡

```
(insert-newconstruct 'set!
     '(set! var term)
     'none
     '((theorem-is 'pre '(subst post var term))))

(insert-newconstruct 'begin
     '(begin *)
     '((rec 'all) (insert-pred-before 'all))
     '((rec 'all)))

(insert-newconstruct 'if
     '((if condition action1)
       (if condition action1 action2))
     '((rec 'action1) (rec 'action2))
     '((rec '((and pre condition) action1 post))
       (rec '((and pre (not condition)) action2 post))))

(insert-newconstruct 'return
     '((return) (return term))
     'none
     '((theorem-is 'pre 'outputspec)))
```

◇
Definition defined by parts 36c, 38b, 39a, 41a.
Definition referenced in part 185a.

The point is that we consider an activity like `(insert-pred-after` *symbol*`)` to be a SCHEME procedure with two arguments called `prog` and `theme`. The argument `theme` states in which of the three stages the generating process actually is: It may have the values `'annotations`, `'generations` and `'guesses`, which correspond to the different `alist`'s of activities. As a consequence `do-activities` only has to look for the kind of the given algorithm, take the activities according to this kind and `theme` and apply these activities to its other argument `prog`:

"kernel.scm" 39b ≡

```
(define (do-activities prog theme)
   (if (is-sequence-without-begin? (car prog))
       (do-activities (cons 'begin prog) theme)
       ⟨get key of prog 183a⟩
       ⟨check format of prog 183b⟩
       (let ((activity-list (get (eval theme) key))
             (ergprog prog))
         (do ((activities activity-list (cdr activities))
              (ergprog prog (if (actual? (car activities) prog)
                                (apply (eval (car activities))
                                       (list ergprog theme))
                                ergprog)))
             ((or (empty? activities)
                  (equal? activities 'none)) ergprog) ))))))
```

◇
File defined by parts 35ab, 39b, 40ab.

The reader may have observed that the activities for recursively annotating algorithms and constructing theorems are both named simply `rec`. We wanted to avoid different names for activities that recursively annotate algorithms resp. construct theorems, so we introduce `rec` as a procedure that calls the "real" procedure according to the given theme:

"kernel.scm" 40a ≡

```
(define (rec symbol)
   (lambda (prog theme)
      (cond ((equal? theme 'annotations)
             ((ann-rec symbol) prog))
            ((equal? theme 'generations)
             ((gen-rec symbol) prog))
            ((equal? theme 'guesses)
             ((guess-rec symbol) prog))
            (else
             (error 'procedure 'rec: theme 'is 'unknown)))))
```

◇
File defined by parts 35ab, 39b, 40ab.

We conclude this section with an easy but powerful activity, namely the `simulate` activity. This activity allows us to reduce the treatment of new constructs to ones already defined: `simulate` is equipped with an abstract algorithm scheme consisting only of already known parts. If called with an algorithm `simulate` constructs a new algorithm according to its abstract scheme and annotates this new one. Thus the new unknown construct is eliminated and for the equivalent part theorems can be constructed using Hoare calculus rules.[1]

"kernel.scm" 40b ≡

```
(define (simulate scheme)
  (lambda (prog . theme)
    (let ((actual-prog (construct prog scheme)))
       (annotate actual-prog))))
```

◇
File defined by parts 35ab, 39b, 40ab.

For example implementing the `repeat`-construct using `simulate` looks as follows. Note that we only need activities for annotating `repeat`. Generating theorems for this construct is done using the activities for the substituted `while`-statement.

⟨insert rules 41a⟩ ≡

```
(insert-newconstruct 'repeat
     '(repeat action until condition)
     '((simulate '(action (while (not condition) action))) ))
```

◇
Definition defined by parts 36c, 38b, 39a, 41a.
Definition referenced in part 185a.

[1] Note that `simulate` allows one to introduce new programming language constructs without taking care of rules to prove their correctness. One only needs to specify an equivalent already processable algorithm.

3.2 Example: Generic Euclidean Algorithm

Before we go into the details of the verification condition generator, we want to illustrate our approach with an example. We consider the generic Euclidean algorithm of section 1.2. Starting with the input file, we describe all three stages of the generator concluding with a verification condition set for the algorithm.

The input file consists of two parts: the algorithm prototype and the algorithm body. The algorithm prototype is a result of the first part of the SuchThat type checker which handles SuchThat declarations: a prototype is the internal counterpart of the algorithm header.

"eucl-procedure.txt" 41b ≡

```
(prototype
   (GCD a b out c)
   (internal (\in u EuclideanRing) (\in v EuclideanRing)
             (\in s EuclideanRing) (\in t EuclideanRing))
   (input (\in a EuclideanRing) (\in b EuclideanRing))
   (output (\in c EuclideanRing)
           (with (\in c Amp) (= c (gcd a b))) ))
```

◇

File defined by parts 41bc.

The second part of the input file is a straightforward translation of the algorithm body into the internal Scheme representation of the SuchThat compiler:

"eucl-procedure.txt" 41c ≡

```
(set! u a)
(set! v b)
(if (= u 0) ((set! c (NF v)) (return)))
(while (not (= v 0))
    ((call QR u v s t)
     (set! u v)
     (set! v t)))
(set! c (NF u))
```

◇

File defined by parts 41bc.

So the input file contains all information about what to prove, namely the pre- and the postcondition and the algorithm itself. But in addition we need the input/output specifications of the subalgorithms to handle procedure calls. We assume their prototypes to be (besides others) in the file `prototypes.txt`.

"prototypes.txt" 42a ≡

```
(prototype
  (QR x y out q r)
  (input (\in x EuclideanRing) (\in y EuclideanRing)
         (with (not (= y 0))))
  (output (\in q EuclideanRing) (\in r EuclideanRing)
          (with (= x (+ (* q y) r))
                (or (= r 0) (< (delta r) (delta y))))))
```

```
    (prototype
      (NF x out y)
      (input (\in x IntegralDomain))
      (output (\in y IntegralDomain)
              (with (\in y Amp) (x is_associated_to y))))
```

◇

File defined by parts 42a, 163.

The first stage of the generator annotates the given algorithm. We use natural numbers i to denote abstract predicates P_i. So, annotating our example results in

"eucl-annotations.txt" 42b ≡

```
    0
    (set! u a) 1
    (set! v b) 2
    (if (= u 0)
        (begin (set! c (NF v)) 3
               (return))) 4
    (while (not (= v 0)) 7
        (begin (call QR u v s t) 6
               (set! u v) 5
               (set! v t))) 8
    (set! c (NF u))
    9
```

◇

Based on these abstract predicates the user can enter loop invariants or any other predicate desired (except for pre- and postcondition (0 and 9) being automatically set to the input and the output specification). Here we only set the loop invariant (7):

```
(put-pred 7 '(= (gcd u v) (gcd a b))).
```

In the second stage Hoare's rules are applied in a backward manner to decompose the original Hoare triple $\{input-specification\}\ algorithm\ \{output-specification\}$ until only programming code free theorems remain. Thereby predicates are still used in an abstract manner only. We get

"eucl-pretheorems.txt" 43 ≡

```
    (implies 0 (subst 1 u a))

    (implies 1 (subst 2 v b))

    (implies (and 2 (= u 0)) (subst true (x y) (v c)))

    (implies (and (and 2 (= u 0))
                  (subst (and (\in y Amp) (x is_associated_to y))
                         (x y) (v c)))
             3)

    (implies 3 (and (\in c Amp) (= c (gcd a b))))

    (implies 4 7)
```

```
    (implies (and 7 (= v 0)) 8)

    (implies (and 7 (not (= v 0))) (subst (not (= y 0)) (x y q r) (u v s t)))

    (implies (and (and 7 (not (= v 0)))
                  (subst (and (= x (+ (* q y) r))
                              (or (= r 0) (< (delta r) (delta y))))
                         (x y q r) (u v s t)))
             6)

    (implies 6 (subst 5 u v))

    (implies 5 (subst 7 v t))

    (implies 8 (subst true (x y) (u c)))

    (implies (and 8 (subst (and (\in y Amp) (x is_associated_to y))
                           (x y) (u c)))
             9)
```

◇

This set of theorems is a set of verification conditions in the sense of [Dil94]: If we find specific counterparts of the abstract predicates that allow proving these theorems, the original algorithm is correct with respect to its specification.

Based on the input/output specification (and the given loop invariant) the last stage of the generator constructs the following nontrivial theorems.

"eucl-theorems.txt" 44 ≡

```
    (implies (and (= u a) (= v b) (= u 0) (\in c Amp) (c is_associated_to b))
             (and (\in c Amp) (= c (gcd a b))))

    (implies (and (= u a) (= v b) (not (= u 0)))
             (= (gcd u v) (gcd a b)))

    (implies (and (= (gcd u v) (gcd a b)) (not (= v 0))
                  (= u (+ (* s v) t)) (or (= t 0) (< (delta t) (delta v))))
             (= (gcd v t) (gcd a b)))

    (implies (and (= (gcd u v) (gcd a b)) (= v 0)
                  (\in c Amp) (c is_associated_to u))
             (and (\in c Amp) (= c (gcd a b))))
```

◇

The first theorem corresponds to step (2), the last one to step (4) of the algorithm. The third theorem states that the formula attached to the loop — predicate 7 from above — indeed is a loop invariant. The second theorem is trivial for the MIZAR proof checker though our trivial-theorem checker has not detected this (and therefore is quite far from being accomplished).

We will prove the first and the two last theorems in chapter five. Thus we will show that the theorems necessary to construct a Hoare calculus derivation for the Euclidean algorithm hold, hence that the generic Euclidean algorithm of section 1.2 is correct with respect to its specification for arbitrary Euclidean domains.

3.3 Annotating Algorithms

An *annotated algorithm* is an algorithm with formulas — known as *annotations* — embedded within it. A *properly annotated algorithm* is an algorithm in which annotations have been inserted at the following points:

(i) before each command γ_i for $0 \leq i < n$ in a sequence of commands $\gamma_1; \gamma_2; \ldots \gamma_n$ and

(ii) after the condition in each loop.

In (i) the sequence $\gamma_1; \gamma_2; \ldots \gamma_n$ must not be a subsequence of a longer sequence of commands. A *properly annotated Hoare triple* is a formula $\{P\}\,\gamma\,\{Q\}$ where γ is a properly annotated algorithm.

Given a properly annotated Hoare triple, it is easy to construct theorems according to the Hoare calculus because each command (with its pre- and postcondition) exactly fits to a backward Hoare rule. Furthermore introducing the intermediate predicates enables the user to take action into the verifying process: He can set as many predicates as he wants.

In the following we describe the activities necessary to carry out annotating algorithms: `insert-pred-after` resp. `insert-pred-before` and `ann-rec`.

Procedure `insert-pred-after` has to handle two cases: Inserting a predicate after each command (in a sequence) and after a special symbol only (for instance after the condition in a *while*-construct).

"annotations.scm" 45a ≡

```
(define (insert-pred-after symbol)
   (lambda (prog . theme)
      (if (equal? symbol 'all)
          ⟨insert-pred-after all 45b⟩
          ⟨insert-pred-after special 46a⟩) ))
```

◇

File defined by parts 45a, 46b, 48b, 49a, 177.

Given the symbol `'all` we begin annotating from the end of the sequence. Note that we distinguish between sequences starting with the key `'begin` or not.

⟨insert-pred-after all 45b⟩ ≡

```
(if (or (empty? prog)
        (empty? (cdr prog)))
    prog
    (if (equal? (car prog) 'begin)
        (let ((rest
               ((insert-pred-after 'all) (cddr prog))))
          (begin
             (set! prednr (+ prednr 1))
             (append (list 'begin (cadr prog))
                     (cons prednr rest))))
```

```
            (let ((rest
                  ((insert-pred-after 'all) (cdr prog))))
              (begin
                 (set! prednr (+ prednr 1))
                 (append (list (car prog) prednr)
                         rest))) ))
```

◇

Definition referenced in part 45a.

Given a special symbol — that is a symbol not equal to `'all` — we look for this symbol in the format definition of the present algorithm. If it exists, we insert a predicate after the corresponding part of the algorithm.

⟨insert-pred-after special 46a⟩ ≡

```
    (do ((format (get formats (get-key prog)) (cdr format))
         (pr prog (cdr pr))
         (ergprog '() (append ergprog (list (car pr)) )))
        ((equal? (car format) symbol)
         (begin
            (set! prednr (+ prednr 1))
            (append (append ergprog (list (car pr)))
                    (cons prednr (cdr pr)) )))
         (if (empty? (cdr format))
             (error 'insert-pred-after: symbol 'does 'not
                    'appear 'in 'format 'of prog)) )
```

◇

Definition referenced in part 45a.

Procedure `insert-pred-before` works in exactly the same way. We omit this procedure here, but it can be found in appendix B.1.

Procedure `ann-rec` has to handle the same cases like `insert-pred-after`:

"annotations.scm" 46b ≡

```
    (define (ann-rec symbol)
       (lambda (prog)
          (if (equal? symbol 'all)
              ⟨ann-rec all 47a⟩
              ⟨ann-rec special 47b⟩) ))
```

◇

File defined by parts 45a, 46b, 48b, 49a, 177.

In the `'all` case — that is the present algorithm is a sequence — we only have to call each part of the sequence recursively.

⟨ann-rec all 47a⟩ ≡

```
    (if (empty? prog)
        prog
        (if (equal? (car prog) 'begin)
            (append (list 'begin (annotate (cadr prog)))
                    ((ann-rec 'all) (cddr prog)))
            (cons (annotate (car prog))
                  ((ann-rec 'all) (cdr prog))) ))
```

◇

Definition referenced in part 46b.

Given a special symbol we look for it in the format definition of the present algorithm and annotate the corresponding part of the algorithm.

⟨ann-rec special 47b⟩ ≡
```
(let* ((key (get-key prog))
       (format (get-actual formats prog)))
  (if (not(member symbol format))
      ⟨check other formats 48a⟩
      (do ((form format (cdr form))
           (pr prog (cdr pr))
           (ergprog '()
                    (append ergprog (list (car pr)) )))
          ((equal? (car form) symbol)
           (append ergprog
                   (append (list (annotate (car pr)))
                           (cdr pr)))) )))
```
◇
Definition referenced in part 46b.

If the given symbol does not appear in the corresponding format, we check whether there are other formats of the present algorithm in which this symbol is included.[1] If not, we report an error.[2]

⟨check other formats 48a⟩ ≡
```
(do ((other-formats (get formats key)
                    (cdr other-formats)))
    ((member symbol (car other-formats)) prog)
  (if (empty? formats)
      (error 'procedure 'rec: symbol 'does 'not
             'appear 'in 'activities 'of prog)))
```
◇
Definition referenced in part 47b.

In the rest of this section we describe the main procedure `make-annotations` for annotating algorithms: We assume that the algorithm is given in a file (written by hand or by the SuchThat compiler) starting with the input/output specification followed by the algorithm as described in the last section. Procedure `make-annotated` first reads the input file (assigning the given algorithm to `proglist` and the prototypes of the file `prototypes.txt` to `spec-list`):

"annotations.scm" 48b ≡
```
(define (make-annotated inputfile outputfile)
   (set! proglist '())
   ⟨read program specs 190c⟩
   ⟨read input file 190b⟩
```
◇
File defined by parts 45a, 46b, 48b, 49a, 177.

[1] for instance the *if*-construct has two formats: with and without alternative.

[2] This is not necessary because `do-activities` checks whether an activity applies to a given algorithm, but it may help to find faulty insertions of programming language constructs.

Before we go into the main loop, we prepare the output file, in which the resulting annotated algorithm is written. Especially we write the first predicate (0) — which stands for the input specification of the algorithm — into the file.

```
"annotations.scm" 49a ≡
    ⟨open output file 191a⟩
    (set! prednr 0)
    (write prednr current-output-port)
    (newline current-output-port)
    ⟨annotate main loop 49b⟩)
◇
```
File defined by parts 45a, 46b, 48b, 49a, 177.

In the main loop each command is annotated and written to the output file. Between two such commands an abstract predicate is inserted.

```
⟨annotate main loop 49b⟩ ≡
    (do ((prog proglist (cdr prog)))
        ((empty? prog) ⟨close output file 191c⟩)
         (begin
            (set! block (annotate (car prog)))
            ⟨write block 191b⟩
            (newline current-output-port)
            (set! prednr (+ prednr 1))
            (write prednr current-output-port)
            (newline current-output-port))) )
◇
```
Definition referenced in part 49a.

3.4 Constructing Abstract Theorems

To construct theorems out of a given Hoare triple we have to apply the rules of Hoare's calculus in a backward manner until the whole program code is replaced. Which rule has to be applied to a special algorithm is given by the activities according to the given algorithm's kind. Here we describe the implementation of these activities: `theorem-is`, `gen-rec` and `is-invariant-for`.

A theorem activity is equipped with two arguments: `assumption` and `conclusion`.[1] These arguments are again abstract schemes (of formulas) that are filled with the corresponding parts of the present algorithm.

```
"theorems.scm" 50a ≡
    (define (theorem-is ass concl)
       (lambda (annotated-prog . theme)
          (if (and (equal? (get-key annotated-prog)
                           'set!)
                   (list? (caddr (cadr annotated-prog))))
              ⟨function call 50b⟩
              (begin
                (set! theorem-list
```

[1] Compare page 29.

```
                    (cons (list 'implies
                                (construct annotated-prog ass)
                                (construct annotated-prog concl))
                          theorem-list))
                annotated-prog) )))
```

◇
File defined by parts 50ac, 51c, 52a.

Procedure calls with only one output variable z are usually written more naturally as $z := f(\vec{x})$ — or `(set! c (f` $\vec{x}$`))` in SCHEME representation. If such a procedure call is detected,[1] we generate theorems for the equivalent call $f(\vec{x}; z)$.

⟨function call 50b⟩ ≡

```
    (begin
      (let ((scheme
             (list 'pre
                   (cons 'call
                         (cons (oper annotated-prog)
                               ((actualparam 'proc) annotated-prog)))
                               'post)))
        (generate-theorems (construct annotated-prog scheme)))
        annotated-prog)
```

◇
Definition referenced in part 50a.

Like the other recursive procedures `gen-rec` has to distinguish between processing sequences and other constructs. Thus we get again

"theorems.scm" 50c ≡

```
    (define (gen-rec scheme)
       (lambda (annotated-prog . theme)
          (if (equal? scheme 'all)
              ⟨gen-rec all 51a⟩
              ⟨gen-rec special 51b⟩) ))
```

◇
File defined by parts 50ac, 51c, 52a.

Given the symbol `'all` we have a sequence. So we generate theorems for each construct of the sequence according to the intermediate predicates (resp. the annotations). We build the current Hoare triple using the already mentioned procedure `construct`.

⟨gen-rec all 51a⟩ ≡

```
    (do ((pr (if (equal? (get-key annotated-prog) 'begin)
                 (append (list (car annotated-prog))
                               (cdadr annotated-prog)
                               (cddr annotated-prog))
                 annotated-prog)
             (cddr pr)))
        ((empty? (cdr pr)) annotated-prog)
         (generate-theorems (construct pr '(pre first intermed))) )
```

◇
Definition referenced in part 50c.

If a special symbol is given, this symbol is an abstract scheme describing which Hoare triple has to be called recursively.[1] So again we construct the triple to be called out of this scheme according to the given algorithm:

[1] Note that both ordinary assignment and this kind of procedure call have `'set!` as key.

[1] In fact this kind of scheme implements Hoare rules not eliminating program code at once.

⟨gen-rec special 51b⟩ ≡

```
(begin
  (generate-theorems (construct annotated-prog scheme))
  annotated-prog)
```

◇

Definition referenced in part 50c.

Procedure `is-invariant-for` is due to our rule for procedure calls requiring the precondition to be invariant for the procedure.[2] Here we only replace abstract parts of this condition according to the given algorithm (using again procedure `construct`) and add it to the sidecondition-list. Checking whether this condition is fulfilled is not possible until specific predicates have been built.

"theorems.scm" 51c ≡

```
(define (is-invariant-for formula proc)
    (lambda (annotated-prog . theme)
       (begin
         (let ((side-cond
                (list 'is-invariant-for
                      (construct annotated-prog formula)
                      ((actualout 'proc) annotated-prog)) ))
           (set! side-cond-list
                 (cons side-cond side-cond-list)))
          annotated-prog)))
```

◇

File defined by parts 50ac, 51c, 52a.

In the main procedure `make-theorems` we again first read an (annotated) algorithm from an input file and prepare the output file. Note, that we do not check syntax of the given algorithm here because we assume the input file to be constructed by the annotation stage of our generator, where formats already have been checked.

"theorems.scm" 52a ≡

```
(define (make-theorems inputfile outputfile)
   (set! proglist '())
   ⟨read input file 190b⟩
   ⟨open output file 191a⟩
   (set! theorem-list '())
   (set! side-cond-list '())
   ⟨make-theorems main loop 52b⟩)
```

◇

File defined by parts 50ac, 51c, 52a.

For each construct of the algorithm (with its corresponding pre- and postcondition) we generate theorems and side conditions and add them to the `theorem-list` resp. `sidecondition-list`:

⟨make-theorems main loop 52b⟩ ≡

```
(do ((prog proglist (cddr prog)))
    ((empty? (cdr prog))
     (begin
        ⟨write+close output file 192a⟩
```

[2] Compare page 29.

```
          (initialize-predlist (+ prednr 1)) ))
        (let ((actual-prog (list (car prog)
                                 (cadr prog)
                                 (caddr prog))))
          (generate-theorems actual-prog))))
```

◇

Definition referenced in part 52a.

3.5 From Abstract to Specific Theorems

Up to this point we have constructed a set of theorems ensuring the correctness of a given algorithm. But all the intermediate predicates that have been introduced according to the rules of the Hoare calculus are still abstract ones. In the following we give some simple rules to fill in this gap. Our examples will show that these easy rules are strong enough to construct theorems for the algorithms we presented in the introduction (provided that the loop invariant for the Euclidean algorithm is given).

Let us first look at the *assignment* construct. If we have $\{P\}\, x := t\, \{Q\}$, we know that setting $P \equiv Q_x[t]$ makes this triple valid. But most algebraic algorithms start with an initialization phase followed by a *loop* or an *if*-statement. To compute specific predicates for a *loop* or an *if*-statement, it is much better to have a specific predicate as precondition. So we decided to implement a different rule that allows — starting with the input-specification — to move forward through the algorithm:[1]

$$\{P\}\ x := t\ \{P \wedge x = t\},\ \text{ if } x \text{ is not free in } P.$$

Note that during the initialization phase the condition is trivially fulfilled. If x is free in P we use the classical rule getting for the most theorems concerning assignment trivial ones.

So we expand our list of rules[2] for computing specific predicates by

```
⟨assignment rule 53⟩ ≡
    (set! guesses
       (put guesses 'set!
            '((set-predicate 'post '(and pre (= var term))
               'provided 'is-not-free 'var 'pre)
              (set-predicate 'pre '(subst post var term)) )))
```

◇

Definition referenced in part 185a.

The next rule concerns the *return*-statement. A `return` stands for a semantic end of the algorithm, so we may assume that such a statement is embraced by an *if*-statement

$$\{P\}\ \texttt{if}\ \text{condition}\ \{A, \texttt{return}\}\ \{Q\}$$

because otherwise the code following the `return` will never be executed. Only if the condition is false, Q will be needed as a precondition for the following statements, so $Q \equiv (P \wedge \neg\text{condition})$ is a reasonable setting. We get

[1] This approach can be compared with the one in [Wan96], where the verification of C++ programs is investigated.

[2] Note that these rules again are considered to be SCHEME procedures.

⟨return rule 54a⟩ ≡

```
(set! guesses
   (put guesses 'if
        '((set-predicate 'post '(and pre (not condition))
           'provided 'is-included 'return 'proc)
          (rec '((and pre condition) action1 post))
          (rec '((and pre (not condition)) action2 post)) )))
```

◇

Definition referenced in part 185a.

Due to the second and the third rule specific theorems are computed for the substatements of the if-construct.

To get specific predicates after a *while*-loop and a procedure call we use rather trivial rules. After the execution of a *while*-loop we know that the loop invariant *inv* as well as the condition's negation hold. So we set $Q \equiv (I \wedge \neg\text{condition})$ getting

⟨while rule 54b⟩ ≡

```
(set! guesses
   (put guesses 'while
        '((set-predicate 'post '(and inv (not condition)))
          (rec '((and inv condition) action inv)) )))
```

◇

Definition referenced in part 185a.

Analogously for the procedure call we have that after executing the call the (invariant) precondition and the subalgorithm's output specification hold:

⟨procedure call rule 54c⟩ ≡

```
(set! guesses
   (put guesses 'call
        '((set-predicate
            'post
            '(and pre
                  (subst (outputspec 'proc)
                         (formalparam 'proc)
                         (actualparam 'proc))) ))))
```

◇

Definition referenced in part 185a.

Note that in both cases this setting leads to at least one trivial theorem of the form $P \longrightarrow P$. In the following we describe the implementation of the activities for constructing specific theorems `set-predicate` and `guess-rec`.

Procedure `set-predicate` has two arguments both being abstract schemes of formulas. The first one is the predicate to be set to the second one. We start with filling in these schemes according to the present algorithm. We do not want to override already defined predicates (especially those having been set by the user), so if the first formula is already specific we do nothing. Otherwise the setting is done, provided that the optional argument `ass` — the condition under which the rule is applicable — can be evaluated to true.

"guesses.scm" 55a ≡

```
(define (set-predicate formula1 formula2 . ass)
   (lambda (annotated-prog . theme)
      (let ((form1 (construct annotated-prog formula1))
            (form2 (construct annotated-prog formula2)))
        (if (is-not-already-specific form1)
            (if (or (empty? ass)
                    (apply (eval (cadr ass))
                           (construct annotated-prog (cddr ass)) ))
                (put-pred form1 form2))))
      annotated-prog))
```

◇

File defined by parts 55ab, 56ac, 178.

Procedure `guess-rec` works exactly like `gen-rec`: To each necessary part of the algorithm the corresponding activities for constructing specific theorems are applied:

"guesses.scm" 55b ≡

```
(define (guess-rec symbol)
   (lambda (annotated-prog . theme)
      (if (equal? symbol 'all)
          (do ((pr (if (equal? (get-key annotated-prog) 'begin)
                       (append (list (car annotated-prog))
                               (cdadr annotated-prog)
                               (cddr annotated-prog))
                       annotated-prog)
                   (cddr pr)))
              ((empty? (cdr pr)) annotated-prog)
               (guess (construct pr '(pre first intermed))) )
          (begin
            (guess (construct annotated-prog symbol))
            annotated-prog))))
```

◇

File defined by parts 55ab, 56ac, 178.

The main procedure for constructing specific theorems starts with setting the first predicate to the input-specification and setting the last to the output-specification. After the main loop — where the above presented rules are applied — it reads the abstract theorems from the input file and fills them in using the just computed specific predicates. Finally, the resulting theorems are written into the output file.

"guesses.scm" 56a ≡

```
(define (make-guesses inputfile outputfile)
   (put-pred 0 ((inputspec) 'proc))
   (put-pred prednr ((outputspec) 'proc))
   ⟨make-guesses main loop 56b⟩
   (set! proglist '())
   ⟨read input file 190b⟩
   ⟨write theorems 192b⟩)
```

◇

File defined by parts 55ab, 56ac, 178.

In its loop, `make-guesses` simply applies procedure `guess` to each part of the given algorithm to get specific predicates necessary to fill in the abstract theorems.

⟨make-guesses main loop 56b⟩ ≡

```
(do ((prog proglist (cddr prog)))
    ((empty? (cdr prog)) ⟨guesses message 193a⟩)
     (let ((actual-prog (list (car prog)
                              (cadr prog)
                              (caddr prog))))
       (guess actual-prog)))
```

◇

Definition referenced in part 56a.

One drawback of the Hoare calculus is the large number of theorems that are constructed, many of them in addition being trivial. So we have included a procedure `make-nontrivial-theorems` that tries to filter out such trivial theorems making the resulting file as short as possible.

"guesses.scm" 56c ≡

```
(define (make-nontrivial-theorems inputfile outputfile)
   (set! proglist '())
   ⟨read input file 190b⟩
   ⟨open output file 191a⟩
   ⟨handle predicates 193b⟩
   (do ((theorems proglist (cdr theorems)))
       ((empty? theorems) ⟨close output file 191c⟩)
        (if (not(is-trivial (car theorems)))
            (begin
              (write (car theorems) current-output-port)
              (newline current-output-port)
              (newline current-output-port))) )))
```

◇

File defined by parts 55ab, 56ac, 178.

This completes the description of our verification condition generator although many other functions exist. These can be found in appendix B; some of the more important ones are commented in appendix B.1.

Chapter 4

Verification of Generic Brown/Henrici Addition

In this chapter we present a machine assisted proof of the correctness of the generic Brown/Henrici addition algorithm of section 1.3. To be more precise, we show that the verification conditions — constructed by the generator of the last chapter — hold by giving the corresponding MIZAR proofs. These verification conditions are given in section 4.1.

To prove these theorems, we need some preparation: We have to provide in MIZAR the necessary algebraic structures and concepts. In doing so, we start from integral domains, which are already included in the MIZAR library. We define ample sets for integral domains and gcd domains in section 4.2 and 4.3 respectively. Furthermore we introduce the concept of normal forms — closely related to ample sets —in section 4.2.

Subsequently we define fractions over an integral domain — the elements the algorithm deals with. Finally, in section 4.5 we prove some exemplary verification conditions, the remaining ones being in appendix A.6.

4.1 Verification Conditions

The output file constructed by the verification condition generator contains 15 nontrivial theorems. The remaining 29 theorems were found to be trivial by the generator. These 15 theorems are divided into two groups: First, there are theorems directly connected to the algorithm's output; that is, theorems stating that the result of the algorithm fulfills its output specification.

Here we only present the theorems we prove in section 4.5. The remaining verification conditions can be found in appendix A.5. Note that ~ stands for the association relation over fractions whereas `is_associated_to` used above demotes the association relation over integral domains based on divisibility.

"BrHenAdd-theorems.txt" 59a ≡

```
(implies (and (is_normalized_wrt r Amp) (is_normalized_wrt s Amp)
          (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
          (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
          (= r2 1) (= s2 1) (= t (fract (+ r1 s1) 1)))
         (and (~ t (+ r s)) (is_normalized_wrt t Amp)))
```

```
    (implies (and (is_normalized_wrt r Amp) (is_normalized_wrt s Amp)
              (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
              (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
              (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
              (\in d Amp) (= d (gcd r2 s2)) (= d 1)
              (= t (fract (+ (* r1 s2) (* r2 s1)) (* r2 s2))))
             (and (~ t (+ r s)) (is_normalized_wrt t Amp)))
```

◇
File defined by parts 59ab, 161, 162.

The first theorem corresponds to the `return` in step `(3)` of the algorithm, the second one to the `return` in step `(5)`, where `d = gcd(r2,s2) = 1`.

The second group consists of theorems concerned with procedure calls. They ensure that at the point where subalgorithms are called, the corresponding input specification is in fact fulfilled. Here, these theorems concern the calls of `fract` and `/`. Again we give two theorems as examples, the remaining ones being listed in appendix A.5.

"BrHenAdd-theorems.txt" 59b ≡

```
    (implies (and (is_normalized_wrt r Amp) (is_normalized_wrt s Amp)
              (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
              (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
              (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
              (\in d Amp) (= d (gcd r2 s2)) (not (= d 1))
              (= r2' (/ r2 d)) (= s2' (/ s2 d))
              (= t1 (+ (* r1 s2') (* s1 r2'))) (= t2 (* r2 s2'))
              (not (= t1 0)) (\in e Amp) (= e (gcd t1 d))
              (= t1' (/ t1 e)))
             (and (not (= e 0)) (e divides t2)))

    (implies (and (is_normalized_wrt r Amp)  (is_normalized_wrt s Amp)
              (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
              (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
              (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
              (\in d Amp) (= d (gcd r2 s2)) (not (= d 1))
              (= r2' (/ r2 d)) (= s2' (/ s2 d))
              (= t1 (+ (* r1 s2') (* s1 r2'))) (= t2 (* r2 s2'))
              (not (= t1 0)) (\in e Amp) (= e (gcd t1 d))
              (= t1' (/ t1 e)) (= t2' (/ t2 e)))
             (not (= t2' 0)))
```

◇
File defined by parts 59ab, 161, 162.

The first theorem establishes the applicability of procedure `/` to compute `t2' = t2/e` in step `(5)` of the algorithm, the other one the applicability of `fract` to compute `t = fract(t1',t2')` at the end of the algorithm.

4.2 Definition of Ample Sets

Ample sets are sets of representatives modulo an equivalence relation (see [Col74] or [Mus71]). Here we need ample sets for the association relation over integral domains. They are necessary for the generic Brown/Henrici addition algorithm in order to get a unique greatest common divisor function. Here is the MIZAR definition:

⟨Definition of AmpleSet 60a⟩ ≡

```
definition
let I be domRing;
mode AmpSet of I -> non empty Subset of the carrier of I means :Def8a:
  (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 of AmpSet 61, ... ⟩

◇

Definition defined by parts 60a, 69b.
Definition referenced in part 138.

The existence of ample sets is due to the axiom of choice. It allows one to chose one element out of each equivalence class of associates of the integral domain. Here we only give the definition of the association classes; correctness proofs and some further properties are included in appendix A.2.

⟨Definition of association classes 60b⟩ ≡

```
definition
let I be domRing;
let a be Element of the carrier of I;
  func Class a -> non empty Subset of the carrier of I means :Defh1:
  (for b being Element of the carrier of I holds
   b ∈ it iff b is_associated_to a);
```

⟨correctness proof of Class 135⟩

```
definition
let I be domRing;
  func Classes I -> Subset-Family of the carrier of I means :Defh2:
  (for A being Subset of the carrier of I holds
   A ∈ it iff (ex a being Element of the carrier of I st A = Class a));
```

⟨correctness proof of Classes 137⟩

◇

Definition referenced in part 134.

We start the existence proof concerning ample sets by setting `M` to the set of association classes of the integral domain `I`. As should be clear, the first goal is to apply the axiom of choice to `M`.

⟨existence proof of AmpSet 61⟩ ≡

```
existence
proof
set M = Classes I;
K1: M is non empty by CL2;
    :: CL2 states that Classes I is non empty (see appendix A.2).
reconsider M as non empty set by K1;
```

◇

Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

Note that in MIZAR the axiom of choice is not an axiom but a theorem[1] contained in the MIZAR article `WELLORD2`. It looks as follows.

[1] It can be proved by use of the axiom of Tarski, which is included in the axiomatics of MIZAR (see the introduction of chapter two).

```
(for X st X ∈ M holds X <> ∅) &
(for X,Y st X ∈ M & Y ∈ M & X <> Y holds X ∩ Y = ∅)
implies
ex Choice being set st
for X st X ∈ M ex x st Choice ∩ X =  x ;
```

To apply this theorem we first have to establish two preconditions about the set of association classes `M`; to be more precise, we have to show that the the association classes `X` form a partition on `M`:

⟨existence proof of AmpSet 62a⟩ ≡

```
K2: for X st X ∈ M holds X <> ∅
```
⟨proof of K2 65a⟩
```
K3: for X,Y st X ∈ M & Y ∈ M & X <> Y holds X ∩ Y = ∅
```
⟨proof of K3 65b⟩

◇
Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

Using `K2` and `K3` we can establish the existence of a set `AmpS'` that contains exactly one element out of each association class `X`. It is easy to prove that this set is nonempty.[1]

⟨existence proof of AmpSet 62b⟩ ≡

```
consider AmpS' being set such that
K5: for X st X ∈ M ex x being Any
    st AmpS' ∩ X = {x} by K2,K3,WELLORD2:27;
K5a: AmpS' is non empty
```
⟨proof of K5a 65c⟩
```
reconsider AmpS' as non empty set by K5a;
```

◇
Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

Unfortunately, so far we cannot conclude that there are no other elements in `AmpS'` but the ones of the integral domain `I`, hence we cannot prove that `AmpS'` $\subseteq$ `I`. So we define a second set `AmpS` containing only those elements of `AmpS'` that are also members of an association class `X`:

⟨existence proof of AmpSet 62c⟩ ≡

```
set AmpS = { x where x is Element of AmpS':
             ex X being non empty Subset of the carrier of I
             st X ∈ M & AmpS' ∩ X = {x}};
```

◇
Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

Note that the properties of `AmpS'` do not automatically carry over to this new set. We have to show again the choice set property for `AmpS`. Subsequently we succeed in proving that `AmpS` is a nonempty subset of `I`, thus that `AmpS` has the type required by our definition.

[1] We need non-emptiness of `AmpS'` introduced by `reconsider` for mode `Element of AmpS'`, which is not accepted by the MIZAR checker for empty sets.

⟨existence proof of AmpSet 63a⟩ ≡

```
K6a: for X being Element of M holds
     ex z being Element of AmpS st AmpS ∩ X = {z}
     ⟨proof of K6a 66a, ... ⟩
K6: AmpS is non empty Subset of the carrier of I
    ⟨proof of K6 68b⟩
reconsider AmpS as non empty Subset of the carrier of I by K6;
```

◇
Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

It remains to show that AmpS has the desired properties, namely that for every element a ∈ I there is an element z ∈ AmpS, such that a and z are associates of each other, and that two distinct elements of AmpS are not associated to each other. In the MIZAR language it looks like this:

⟨existence proof of AmpSet 63b⟩ ≡

```
K7: for a being Element of the carrier of I
    ex z being Element of AmpS st z is_associated_to a
    ⟨proof of K7 63c⟩
K8: for x,y being Element of AmpS holds
    x <> y implies x is_not_associated_to y
    ⟨proof of K8 64a, ... ⟩
thus thesis by K7,K8;
end;
end;
```

◇
Definition defined by parts 61, 62abc, 63ab.
Definition referenced in part 60a.

Properties K7 and K8 of course hold because AmpS is defined via the axiom of choice. K7 is an immediate consequence of the fact that for each subset of M there is an element z of this subset being also a member of AmpS.

⟨proof of K7 63c⟩ ≡

```
proof
let a be Element of the carrier of I;
H0: Class a ∈ M by Defh2;  :: remember that M = Classes I
reconsider N = Class a as Element of M by H0;
consider z being Element of AmpS such that
H1: AmpS ∩ N = {z} by K6a;
H1a: z ∈ {z} by ENUMSET1:4;
H1b: z ∈ AmpS ∩ Class a by H1a,H1;
H2: z ∈ Class a by H1b,BOOLE:def 3;
H3: z is_associated_to a by H2,Defh1;
thus thesis by H3;
end;
```

◇
Definition referenced in part 63b.

Property K8 follows because there is only one element out of each association class in AmpS. We proceed by contradiction, first proving that both x and y are contained in AmpS ∩ Class x if they are Elements of AmpS and associates of each other.

⟨proof of K8 64a⟩ ≡

```
proof
let x,y be Element of AmpS;
assume H0: x <> y;
assume H1: x is_associated_to y;
H2: x is_associated_to x & y is_associated_to x by H1,L2;
H3: x ∈ Class x & y ∈ Class x by H2,Defh1;
H6: x ∈ AmpS ∩ Class x & y ∈ AmpS ∩ Class x by H3,BOOLE:def 3;
```

◇
Definition defined by parts 64ab.
Definition referenced in part 63b.

We know — by the axiom of choice — that `AmpS ∩ Class x` is a one-element set. Consequently we get `x = y` by using `H6`, a contradiction.

⟨proof of K8 64b⟩ ≡

```
H8: Class x ∈ M by Defh2;
consider z being Element of AmpS such that
H9: AmpS ∩ Class x = {z} by H8,K6a;
H10: x ∈ {z} & y ∈ {z} by H6,H9;
H11: x = z & y = z by H10,ENUMSET1:3;
thus thesis by H0,H11;
end;
```

◇
Definition defined by parts 64ab.
Definition referenced in part 63b.

This completes the main level of the existence proof concerning ample sets. In the following we want to fill in the gaps we left in order to make clear the overall structure of the proof. The reader not being interested in these details may continue at page 55.

We start with proving properties `K2` and `K3` about `M = Classes I` that we needed above in order to apply the axiom of choice at level `K5`. The first one states, that each subset `X` of `M` is nonempty, which is easy to prove because each `X` is an association class of the integral domain `I`.

⟨proof of K2 65a⟩ ≡

```
proof
let X be Any such that H0: X ∈ M;
consider A being Element of the carrier of I such that
H1: X = Class A by H0,Defh2;
thus thesis by H1;
end;
```

◇
Definition referenced in part 62a.

The second condition requires that two subsets `X` and `Y` are either equal or distinct, thus that the association classes induce an equivalence relation on `I`. This is an easy exercise we proved in theorem `CL1`. We do not present the proof of `CL1` here, but it is included in appendix A.2.

⟨proof of K3 65b⟩ ≡

```
proof
let X,Y be Any such that H0: X ∈ M & Y ∈ M & X <> Y;
assume H1: X ∩ Y <> ∅;
consider A being Element of the carrier of I such that
H2: X = Class A by H0,Defh2;
consider B being Element of the carrier of I such that
H3: Y = Class B by H0,Defh2;
H4: X = Y by H1,H2,H3,CL1;
thus contradiction by H0,H4;
end;
```

◇
Definition referenced in part 62a.

At level K5a we had to show that the set AmpS' defined by applying the axiom of choice is nonempty. This follows by taking an arbitrary association class: Due to the definition of AmpS', there is an element x out of this class being also in AmpS'.

⟨proof of K5a 65c⟩ ≡

```
proof
M0: Class 1.I ∈ M by Defh2;
consider x being Any such that
M1: AmpS' ∩ Class 1.I = {x} by K5,M0;
M2: x ∈ {x} by ENUMSET1:4;
M3: x ∈ AmpS' ∩ Class 1.I by M2,M1;
thus thesis by M3,BOOLE:def 3;
end;
```

◇
Definition referenced in part 62b.

As we said above, the properties of AmpS' do not automatically carry over to AmpS. Of course they hold because AmpS is defined via AmpS', but we have to prove the existence of an element x with AmpS ∩ X = {x} again for the new set AmpS. We start by showing that x ∈ AmpS and x ∈ AmpS ∩ X if x is in AmpS ∩ X = {x}.

⟨proof of K6a 66a⟩ ≡

```
proof
let X be Element of M;
consider x being Any such that
H1: AmpS' ∩ X = {x} by K5;
M4: x ∈ AmpS
    ⟨proof of M4 67b⟩
H2b: x ∈ AmpS ∩ X
     ⟨proof of H2b 68a⟩
```

◇
Definition defined by parts 66abc, 67a.
Definition referenced in part 63a.

To establish our assertion AmpS ∩ X = {x}, we have to prove two inclusions. The first one is easy to show, because y ∈ {x} implies y ∈ AmpS ∩ X by the just proven level H2b.

⟨proof of K6a 66b⟩ ≡

```
H3a: for y being Any holds y ∈ {x} implies y ∈ AmpS ∩ X
     by H2b,ENUMSET1:3;
```

◇
Definition defined by parts 66abc, 67a.
Definition referenced in part 63a.

To prove the second implication — AmpS ∩ X ⊆ {x} — we go back to the set AmpS', of which we know that AmpS' ∩ X = {x}. Thus we get the desired thesis by showing that y ∈ AmpS ∩ X implies y ∈ AmpS', which follows by the definition of AmpS.

⟨proof of K6a 66c⟩ ≡

```
H3b: for y being Any holds y ∈ AmpS ∩ X implies y ∈ {x}
     proof
     let y be Any;
     assume M0: y ∈ AmpS ∩ X;
     M1a: y ∈ AmpS & y ∈ X by M0,BOOLE:def 3;
     consider zz being Element of AmpS' such that
     M1: y = zz &
         (ex X being non empty Subset of the carrier of I
          st X ∈ M & AmpS' ∩ X = {zz}) by M1a;
     M2a: y ∈ AmpS' by M1;
     M2: y ∈ AmpS' ∩ X by M1a,M2a,BOOLE:def 3;
     thus thesis by M2,H1;
     end;
```

◇
Definition defined by parts 66abc, 67a.
Definition referenced in part 63a.

Consequently, we can prove the choice set property of AmpS using levels H3a and H3b. Note that we also have to reference label M4 stating that x indeed is an element out of AmpS and not only of type Any.

⟨proof of K6a 67a⟩ ≡

```
H3c: AmpS ∩ X = {x} by H3a,H3b,TARSKI:2;
thus thesis by H3c,M4;
end;
```

◇
Definition defined by parts 66abc, 67a.
Definition referenced in part 63a.

To establish the choice set property of AmpS, all that remains to show is x ∈ AmpS and x ∈ AmpS ∩ X (the proofs of levels M4 and H2b we left out above). The first assertion follows by using the definition of the set AmpS', the second one is an easy consequence of the first:

⟨proof of M4 67b⟩ ≡

```
proof
M0a: X ∈ Classes I;
M0: X is non empty Subset of the carrier of I by M0a,CL3;
M1a: x ∈ {x} by ENUMSET1:4;
M2: x ∈ AmpS' by M1a,H1,BOOLE:def 3;



M3: ex X being non empty Subset of the carrier of I
    st X ∈ M & AmpS' ∩ X = {x} by M0,H1;
thus thesis by M2,M3;
end;
```

◇
Definition referenced in part 66a.

⟨proof of H2b 68a⟩ ≡

```
proof
M1a: x ∈ {x} by ENUMSET1:4;
M1: x ∈ X by M1a,H1,BOOLE:def 3;
thus thesis by M1,M4,BOOLE:def 3;
end;
```

◇

Definition referenced in part 66a.

To fill in the last gap in the existence proof for ample sets of integral domains we have to show that AmpS is a nonempty subset of I This is done in MIZAR as follows:

⟨proof of K6 68b⟩ ≡

```
proof
H0: AmpS is non empty
    ⟨proof of H0 68c⟩
H1: for z being Any holds
    z ∈ AmpS implies z ∈ the carrier of I
    ⟨proof of H1 69a⟩
thus thesis by H0,H1,TARSKI:def 3;
end;
```

◇

Definition referenced in part 63a.

To prove level H0, we proceed the same way, we used to show non-emptiness of the set AmpS': We know that there is an element x being a member of both AmpS' and Class 1.I. Thus, this x also is a member of AmpS by definition.

⟨proof of H0 68c⟩ ≡

```
proof
M0: Class 1.I ∈ M by Defh2;
consider x being Any such that
M1: AmpS' ∩ Class 1.I = {x} by K5,M0;
M2: x ∈ {x} by ENUMSET1:4;
M3: x ∈ AmpS' ∩ Class 1.I by M2,M1;
M4: x ∈ AmpS' by M3,BOOLE:def 3;
M5: x ∈ AmpS by M4,M1,M0;
thus thesis by M5;
end;
```

◇

Definition referenced in part 68b.

The proof of level H1 is a trivial consequence of the definition of AmpS. Remember that the property proved here was the reason for introducing AmpS.

⟨proof of H1 69a⟩ ≡

```
proof
let z be Any;
assume H3: z ∈ AmpS;
consider x being Element of AmpS' such that
H4: z = x &
    (ex X being non empty Subset of the carrier of I
     st X ∈ M & AmpS' ∩ X = {x}) by H3;
consider X being non empty Subset of the carrier of I such that
H4a: X ∈ M & AmpS' ∩ X = {z} by H4;
H5: z ∈ {z} by ENUMSET1:4;
H6: z ∈ AmpS' ∩ X by H4a,H5;
H7: z ∈ X by H6,BOOLE:def 3;
thus thesis by H7;
end;
```

◇

Definition referenced in part 68b.

So far, we established the existence of ample sets for association classes in integral domains. As a matter of convenience we require that `1.I` always is an element of our ample sets — whereas `0.I` anyhow is contained in every ample set, because the association class of `0.I` contains only one element. Consequently, we define:

⟨Definition of AmpleSet 69b⟩ ≡

```
definition
let I be domRing;
mode AmpleSet of I -> non empty Subset of the carrier of I means :Def8:
  it is AmpSet of I &
  1.I ∈ it;
⟨existence proof of AmpleSet 139⟩

reserve Amp for AmpleSet of I;
```

◇
Definition defined by parts 60a, 69b.
Definition referenced in part 138.

It is easy to show the existence of these special ample sets: In an ordinary ample set there is an element `x` being associated to `1.I`. We only have to take an ample set and exchange this `x` by `1.I`:

⟨Defining AmpleSet 69c⟩ ≡

```
let A be AmpSet of I;
consider x being Element of A such that
H1: x is_associated_to (1.I) by Def8a;
set A' = { z where z is Element of A : z <> x } U {(1.I)};
```

◇
Definition referenced in part 139.

The rest of the proof consists of showing that `A'` again fulfills the definition of an ample set for the association classes of the integral domain `I`. It is very close to the proof just given, so we omit it here (it can be found in appendix A.2).

As we will see, using ample sets to define the greatest common divisor function only suffices to show `gcd(num(t),denom(t)) = 1` for the output `t` of the Brown/Henrici addition algorithm. To establish that `t` is a normalized fraction — that is, in addition holds that `denom(t)` is an element of the ample set[1] — we need our ample sets to be multiplicative:

⟨Definition of multiplicative AmpleSet 70a⟩ ≡

```
definition
let I be domRing;
let Amp be AmpleSet of I;
  pred Amp is_multiplicative means :Def25:
  for x,y being Element of Amp holds x*y ∈ Amp;
end;
```

◇
Definition defined by parts 70ab.
Definition referenced in part 142a.

[1] See section 4.4 for a thorough Mizar definition of normalized fractions.

The main property of multiplicative ample set we will use to verify the Brown/Henrici addition algorithm, is that they are also closed with respect to division:

⟨Definition of multiplicative AmpleSet 70b⟩ ≡

```
theorem
for Amp being AmpleSet of I holds
Amp is_multiplicative implies
(for x,y being Element of Amp holds
 (y divides x & y <> (0.I)) implies x/y ∈ Amp)
```

⟨proof of AMP5 142b⟩

◇

Definition defined by parts 70ab.
Definition referenced in part 142a.

We conclude this section with the definition of a normal form modulo an ample set, though this is not necessary for verification of the Brown/Henrici addition algorithm.[2] The normal form of an element `x` out of `I` is nothing more than the element `z` of the corresponding ample set associated to `x`:

⟨Definition of Normal Form 71a⟩ ≡

```
definition
let I be domRing;
let Amp be AmpleSet of I;
let x be Element of the carrier of I;
 func NF(x,Amp) -> Element of the carrier of I means :Def20:
 it ∈ Amp & it is_associated_to x;
```

⟨correctness proof of normal form 144⟩

◇

Definition referenced in part 143.

4.3 Definition of Gcd Domains

A gcd domain is an integral domain I, in which for each two elements x and y in I a greatest common divisor exists. Integral domains are already included in the MIZAR library (where they are called `domRing`). To introduce gcd domains we define the following attribute `gcd-like` on integral domains.

⟨Definition of gcdDomain 71b⟩ ≡

```
definition
let I be domRing;
attr I is gcd-like means :Def7:
  (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));
```

[2] We need the definition when verifying the generic Euclidean algorithm of section 1.2 in chapter 5. Note also that the predicate `normalized` of section 4.4 can be defined via normal forms using the fact that `a ∈ Amp` iff `a = NF(a,Amp)`.

```
    end;
```
◇
Definition defined by parts 71bc.
Definition referenced in part 145a.

In MIZAR each type must have nonempty denotation.[1] Consequently, before defining mode `gcdDomain` as `gcd-like domRing`, we have to show that a `domRing` fulfilling the attribute `gcd-like` exists. Formally this is done with an *existential cluster*. After the cluster definition we succeed in defining mode `gcdDomain` as indicated.

⟨Definition of gcdDomain 71c⟩ ≡
```
    definition
     cluster gcd-like domRing;
    existence
     ⟨existence proof of gcdDomain 72a⟩
    end;

    definition
      mode gcdDomain is gcd-like domRing;
    end;
```
◇
Definition defined by parts 71bc.
Definition referenced in part 145a.

Fortunately we need not prove the existence of gcd domains from scratch. MIZAR already contains the algebraic structure `Field`, so we proceed by proving that a field is a gcd domain. To show that a field is an integral domain, we use the corresponding theorem out of the MIZAR library.[2] It remains to prove that a field is `gcd-like`:

⟨existence proof of gcdDomain 72a⟩ ≡
```
    proof
    consider F being strict Field;
    reconsider F as domRing by VECTSP_2:13;
    H4: F is gcd-like
        ⟨proof of gcd-like 72b⟩
    thus thesis by H4;
    end;
```
◇
Definition referenced in part 71c.

To show that the field `F` is gcd-like, we have to find a greatest common divisor for each pair of elements `x` and `y` out of `F`.[1] We proceed by considering two cases:

⟨proof of gcd-like 72b⟩ ≡
```
    proof
    let x,y be Element of the carrier of F;
    H3: now per cases;
    case A: x <> 0.F;
    ⟨proof of gcd-like, case A 73b⟩
    case B: x = 0.F;
```

[1] This prohibits modes like `non empty empty set`.

[2] Note that the type of `F` has to be changed using `reconsider`, because otherwise MIZAR will not accept step `H4`. The reason is that attribute `gcd-like` (as well as `divides`) is defined for `domRing` only.

[1] Note that in fields for x and y not both being zero every element $z \neq 0$ is a greatest common divisor of x and y. Using in addition ample sets for normalization, this implies that 1 and 0 are the only possible values for greatest common divisors in fields.

```
    ⟨proof of gcd-like, case B 73a⟩
    thus thesis by H3;
    end;
```
◇
Definition referenced in part 72a.

We start with case B: If x = 0.F, then y is a greatest common divisor of x and y. The proof is simply done by listing the required properties properties of the attribute gcd-like.

```
⟨proof of gcd-like, case B 73a⟩ ≡
    B0: y divides y by L1;
    B1: y*0.F = 0.F by VECTSP_2:26;
    B2: y divides 0.F by B1,Def1;
    B3: for z being Element of the carrier of F
        st (z divides 0.F & z divides y)
        holds z divides y;
    thus thesis by B,B0,B2,B3;
    end;
```
◇
Definition referenced in part 72b.

To show the other case — x ≠ 0.F —, we prove that 1.F is a greatest common divisor of x and y. The first two properties of gcd-like — 1.F divides x and 1.F divides x — are an immediate consequence of 1.F being the multiplicative identity of field F.

```
⟨proof of gcd-like, case A 73b⟩ ≡
    A1: x = 1.F*x & y = 1.F*y by VECTSP_2:1;
    A2: 1.F divides x & 1.F divides y by A1,Def1;
    A5: for z being Element of the carrier of F
        st (z divides x & z divides y)
        holds z divides 1.F
        ⟨proof of gcd-like, case A, label A5 73c, ... ⟩
    thus thesis by A2,A5;
```
◇
Definition referenced in part 72b.

It remains to show that every element z dividing both x and y also divides 1.F. If we have z ≠ 0.F, this follows by taking the multiplicative inverse z' of z.

```
⟨proof of gcd-like, case A, label A5 73c⟩ ≡
    proof
    let z be Element of the carrier of F;
    M1: now per cases;
    case A1: z <> 0.F;
    consider z' being Element of the carrier of F such that
    M11: z*z' = 1.F by A1,VECTSP_1:def 20;
    thus z divides 1.F by M11,Def1;
```
◇
Definition defined by parts 73c, 74.
Definition referenced in part 73b.

If z = 0.F — and z divides x —, we conclude that also x equals 0.F — a contradiction to the assumption x ≠ 0.F from above:

```
⟨proof of gcd-like, case A, label A5 74⟩ ≡
    case A2: z = 0.F;
```

```
    assume M12: z divides x;
    consider d being Element of the carrier of F such that
    M13: 0.F*d = x by M12,Def1,A2;
    M14: x = 0.F by M13,VECTSP_2:26;
    thus z divides 1.F by M14,A;
    end;  :: cases
    thus thesis by M1;
    end;
```

◇

Definition defined by parts 73c, 74.
Definition referenced in part 73b.

So far, we established the existence of gcd domains (the definition of the greatest common divisor function can be found in section 2.3) and ample sets for integral domains in MIZAR. In the next section we start with the actual correctness proof of the generic Brown/Henrici addition algorithm by introducing the algebraic structures and objects the algorithm deals with.

4.4 Definition of Fractions

In this section we present MIZAR definitions for the domains and functions the generic Brown/Henrici addition algorithm works on. We have to introduce fractions over an integral domain as well as constructors for them: `num`, `denom` and `fract`. Also we need addition of two fractions, additive and multiplicative unity of fractions and two further predicates `~` and `is_normalized_wrt`.

The above mentioned definitions are contained in the MIZAR article `BrHenAdd.miz`, which of course begins with the necessary environment.

"BrHenAdd.miz" 75a ≡

⟨BrHenAdd environment 164⟩

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We start with the introduction of fractions over an integral domain `I`. A fraction is a pair over `I` with its second element being not zero. Consequently we use the MIZAR constructors for arbitrary pairs, defined in the article `MCART_1`. Existence of fractions is simply proved by showing that `[0,1]` is of this kind.

"BrHenAdd.miz" 75b ≡

```
definition
let I be domRing;
mode Fraction of I
        -> Element of [:the carrier of I,the carrier of I:]
means :Def52:
 ex a,b being Element of the carrier of I
 st (it = [a,b] & b <> 0.I);
existence
 proof
 H1: 1.I <> 0.I by VECTSP_1:def 21;
 take [0.I,1.I];
 thus thesis by H1;
 end;
end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The set `Q` of fractions over an integral domain `I`[1] contains all just defined pairs. In the MIZAR language this is described by saying `u ∈ Q` if and only if a suitable predicate over `u` holds.

"BrHenAdd.miz" 76a ≡

```
definition
let I be domRing;
mode Fractions of I
       -> non empty Subset of [:the carrier of I,the carrier of I:]
means :Def57:
```

[1] Compare the global declarations of the algorithm on page 6.

```
 for u being Any holds
 u ∈ it iff ex a,b being Element of the carrier of I st
            (u = [a,b] & b <> 0.I);
```
⟨existence proof of fractions 76b⟩

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Existence of the set `Q` is shown by just taking the desired set (here called `M`). Note that the main effort goes into proving that `M` has the type required by the definition, that is in proving that `M` in fact is a nonempty subset of the Cartesian product of the integral domain `I` — whereas the defining property of `Q` trivially holds for `M`.

⟨existence proof of fractions 76b⟩ ≡

```
existence
proof
set M = {[a,b] where a,b is Element of the carrier of I:
               b <> 0.I };
H2: for u being Any holds
    u ∈ M iff ex a,b being Element of the carrier of I st
              (u = [a,b] & b <> 0.I);
H0: for u being Any holds u ∈ M implies
    u ∈ [:the carrier of I,the carrier of I:]
    proof
    let u be Any;
    assume H12: u ∈ M;
    H13: ex a,b being Element of the carrier of I st
        (u = [a,b] & b <> 0.I) by H12;
    thus thesis by H13;
    end;
H1: M is Subset of [:the carrier of I,the carrier of I:]
    by H0,TARSKI:def 3;
H3: M is non empty
    proof
    H31: 1.I <> 0.I by VECTSP_1:def 21;
    consider u being Any such that H32: u = [0.I,1.I];
    H33: u ∈ M by H31,H32;
    thus thesis by H33;
    end;
thus thesis by H1,H2,H3;
end;
end;
```

◇

Definition referenced in part 76a.

In the following we show two simple consequences of our definitions, nevertheless being very helpful in later proofs. The first one states that for every fraction the second element of the corresponding pair — the denominator — is not zero.

"BrHenAdd.miz" 77a ≡

```
theorem
N:for I being domRing
for u being Fraction of I holds u`2 <> 0.I
```

```
    proof
    let I be domRing;
    let u be Fraction of I;
    consider a,b being Element of the carrier of I such that
    H0: u = [a,b] & b <> 0.I by Def52;
    H1: u`2 = b by H0,MCART_1:def 2;
    thus thesis by H0,H1;
    end;
```
◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The second property connects the set `Q` of fractions with the individual fractions `u`. Note that the proof is nothing more than referencing the two definitions.

"BrHenAdd.miz" 77b ≡
```
    theorem
    for I being domRing
    for Q being Fractions of I
    for u being Fraction of I holds u is Element of Q
    proof
    let I be domRing;
    let Q be Fractions of I;
    let u be Fraction of I;
    H0: ex a,b being Element of the carrier of I
        st (u = [a,b] & b <> 0.I) by Def52;
    thus thesis by H0,Def57;
    end;
```
◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Now that we have made fractions available in MIZAR, we define constructors for them: the functions `num`, `denom` and `fract`. This is not necessary — we could use the above mentioned pair constructors of MIZAR — but we prefer having the same vocabulary in the algorithm and its corresponding MIZAR proof. Consequently the following definitions are hardly more than renaming the pair constructors of MIZAR for the special case of fractions over integral domains `I` (this includes that the type of the result is — contrary to the one of arbitrary pairs — `Element of the carrier of I`).

"BrHenAdd.miz" 78 ≡
```
    definition
    let I be domRing;
    let u be Fraction of I;
    func num(u) -> Element of the carrier of I means :Def55:
     it = u`1;
    correctness;
    end;

    definition
    let I be domRing;
    let u be Fraction of I;
    func denom(u) -> Element of the carrier of I means :Def53:
     it = u`2;
```

```
    correctness;
    end;

    definition
    let I be domRing;
    let u1,u2 be Element of the carrier of I;
    assume A:u2 <> 0.I;
    func fract(u1,u2) -> Fraction of I means :Def54:
     it = [u1,u2];
    existence
     proof
     consider u being Any such that H3: u = [u1,u2];
     H1: u is Fraction of I by H3,A,Def52;
     thus thesis by H1,H3;
     end;
    uniqueness;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Before going on with defining addition of fractions, we want to prove three theorems about the just defined functions `num`, `denom` and `fract`. The proofs are simple (they require application of definitions only), so we omit them here. The interested reader can find them in appendix A.4.

The first theorem shows that the usual defining equation about our constructors holds:

"BrHenAdd.miz" 79a ≡

```
    theorem
    for I being domRing
    for u being Fraction of I holds u = fract(num(u),denom(u))
    ⟨proof of fraction's constructor equation 156b⟩
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The second one shows that for fractions `u` the denominator `denom(u)` is not zero. It is just a reformulation of theorem `N` on page 61, which stated that the second pair element `u‘2` does not equal zero.

"BrHenAdd.miz" 79b ≡

```
    theorem
    TT: for I being domRing
    for u being Fraction of I holds
    denom(u) <> 0.I
    ⟨proof of denom 156c⟩
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The last theorem connects fractions with the corresponding elements of the integral domain `I`. We need this property later in our verification proofs, where we have to decompose fractions into elements of the integral domain, to apply the theorem of Brown and Henrici we proved in section 2.3.

"BrHenAdd.miz" 80a ≡

```
    theorem
    F1:for I being domRing
    for u being Fraction of I
    for a being Element of the carrier of I
    for b being Element of the carrier of I st b <> 0.I holds
    (a = num(u) & b = denom(u)) iff fract(a,b) = u
```

⟨proof of F1 157a⟩

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

In the following we introduce addition of fractions[1] over an integral domain `I`. We use the usual definition giving not the representatives of the equivalence relation over fractions (see below):

$$\mathtt{a/b} + \mathtt{c/d} := (\mathtt{ad} + \mathtt{bc})/\mathtt{bd}.$$

Again the existence proof is easy: It is trivial to get a fraction `u` with the desired properties; all we have to show is that this `u` is of the type required by the definition, namely that it is a pair over the integral domain:

"BrHenAdd.miz" 80b ≡

```
    definition
    let I be domRing;
    let u,v be Fraction of I;
    func u+v -> Fraction of I means :Def70:
     it = [u‘1*v‘2+v‘1*u‘2, u‘2*v‘2];
    existence
     proof
     H1: u‘2 <> 0.I & v‘2 <> 0.I by N;
     H2: u‘2*v‘2 <> 0.I by H1,VECTSP_2:15;
     consider a being Element of the carrier of I such that
     H6: a = u‘1*v‘2+v‘1*u‘2;
     consider b being Element of the carrier of I such that
     H7: b = u‘2*v‘2;
     consider u being Element of [:the carrier of I,the carrier of I:]
     such that H3: u = [a,b];
     H4: u is Fraction of I by H3,H2,H7,Def52;
     thus thesis by H3,H4,H6,H7;
     end;
    uniqueness;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Note that this definition implies that the result `t` of the Brown/Henrici addition algorithm in general does not fulfill `t = r+s`. To overcome this problem, it is possible to define the sum `u+v` by `(u+v)‘1 * u‘2*v‘2 = (u+v)‘2 * u‘1*v‘2+v‘1*u‘2`, thus stating that `u+v` and `[u‘1*v‘2+v‘1*u‘2, u‘2*v‘2]` belong to the same equivalence class. In this case the sum `u+v` is not uniquely determined, hence we would have to define `u+v` in MIZAR not as a function, but as a mode. However we prefer defining `+` as a function, thus being forced to accept that not `t = r+s`, but only `t ~ r+s` — where `~` stands for the usual equivalence relation over fractions — holds for the result `t` of Brown/Henrici addition.

[1] Note that it is no problem in MIZAR to use the symbol `+` to denote addition over both the integral domain `I` and fractions over `I`.

The following theorem[1] classifies addition of fractions in terms of our constructors `num`, `denom` and `fract`.

"BrHenAdd.miz" 81a ≡

```
theorem
for I being domRing
for u,v being Fraction of I holds
u+v = fract(num(u)*denom(v)+num(v)*denom(u),denom(u)*denom(v))
```
⟨proof of fraction addition 158a⟩

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Again we present a theorem that states a connection between addition of fractions and the corresponding elements of the integral domain `I`. The proof is done by just substituting the elements of `I` standing for numerators and denominators in the definition of the addition function.

"BrHenAdd.miz" 81b ≡

```
theorem
F2:for I being domRing
for r,s being Fraction of I
for r1,r2,s1,s2 being Element of the carrier of I holds
(r1 = num(r) & r2 = denom(r) & s1 = num(s) & s2 = denom(s)) implies
num(r+s) = r1*s2+s1*r2 & denom(r+s) = r2*s2
```
⟨proof of F2 157b⟩

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We carry on with defining the additive and multiplicative unity `0.Q` and `1.Q` for the set of fractions `Q` over an integral domain. This requires nothing more than renaming `fract(0.I,1.I)` and `fract(0.I,1.I)` respectively. Consequently the corresponding correctness proofs are trivial. Theorems showing that `0.Q` and `1.Q` indeed are unities for fractions can be found in appendix A.4.[2]

"BrHenAdd.miz" 82a ≡

```
definition
let I be domRing;
let Q be Fractions of I;
func 0.Q -> Fraction of I means :Def74:

 it = fract(0.I,1.I);
correctness;
end;

definition
let I be domRing;
let Q be Fractions of I;
```

[1] The MIZAR proofs of this and the next theorem can be found in appendix A.4.

[2] For completion, we also defined multiplication of fractions, although we did not need this operation for the verification of Brown/Henrici addition. (It is necessary to prove generic Brown/Henrici multiplication correct; see [Sch97b].)

```
    func 1.Q -> Fraction of I means :Def75:
     it = fract(1.I,1.I);
    correctness;
    end;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

It remains to define the above mentioned predicates ~ and `is_normalized_wrt`. They are necessary to express the input/output specification of Brown/Henrici addition in MIZAR: Proving the algorithm correct consists mainly of showing that the output `t` fulfills these two predicates.

The first one describes, when two fractions belong to the same equivalence class; that is, when they denote the same value.

"BrHenAdd.miz" 82b ≡
```
    definition
    let I be domRing;
    let u,v be Fraction of I;
    pred u ~ v means :Def76:
     num(u)*denom(v) = num(v)*denom(u);
    end;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The other predicate is the already several times mentioned `is_normalized_wrt`. Note that it can be defined over gcd domains only. In fact this is the reason for the Brown/Henrici algorithm computing in gcd domains only, and not in arbitrary integral domains.

"BrHenAdd.miz" 83 ≡
```
    definition
    let G be gcdDomain;
    let u be Fraction of G;
    let Amp be AmpleSet of G;
    pred u is_normalized_wrt Amp means :Def73:
      gcd(num(u),denom(u),Amp) = 1.G &
      denom(u) ∈ Amp;
    end;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

So far, we finished the necessary preparations to prove the verification condition for the generic Brown/Henrici addition algorithm, which follows next.

4.5 Proving the Verification Conditions

In this section we show, how to prove verification conditions for the generic Brown/Henrici addition algorithm — and consequently the correctness of this algorithm — using MIZAR. Here we only present four exemplary theorems, the ones given in section 4.1. The remaining proofs are included in appendix A.6.

We start with MIZAR reservations for the necessary algebraic objects. Note that these reservations directly correspond to the global and local declarations of the Brown/Henrici addition algorithm.

"BrHenAdd.miz" 84a ≡

```
reserve G for gcdDomain;
reserve Q for Fractions of G;
reserve Amp for AmpleSet of G;
reserve s,r,t for Fraction of G;
reserve r1,r2,s1,s2,d,e,r2',s2',t1,t2,t1',t2'
             for Element of the carrier of G;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

In the first part of this section we prove two theorems about the correctness of procedure calls. The first one states correctness of the application of procedure / to compute `t2' = t2/e` in step (5) of the algorithm. Note that the following is a direct translation of the theorem in SCHEME representation constructed by our verification condition generator.

"BrHenAdd.miz" 84b ≡

```
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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
 r2' = r2/d & s2' = s2/d &
 t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
 t1 <> 0.G & e ∈ Amp & e = gcd(t1,d,Amp) & t1' = t1/e)
implies (e <> 0.G & e divides t2)
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

As obvious the proof starts with stating the assumptions. Note that we do not need to list all assumptions of the theorem, but only those necessary to prove the assertions.

"BrHenAdd.miz" 85a ≡

```
    proof
    M: now assume
    H0: d = gcd(r2,s2,Amp) & t2 = r2*s2' &
        t1 <> 0.G & e = gcd(t1,d,Amp);
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

To show `e divides t2 = r2*s2'` we use the definition of the greatest common divisor function and transitivity of `divides`, which was proved in theorem `GCD:2`.[1]

"BrHenAdd.miz" 85b ≡

```
    H1: e divides d & d divides r2 by H0,GCD:def 12;
    H4: e divides r2 by H1,GCD:2;
    H5: e divides r2*s2' by H4,GCD:7;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The remaining `e` $\neq$ `0.G` is an immediate consequence of theorem `GCD:33`, stating that the greatest common divisor of `a` and `b` is zero if and only if `a = 0.G` and `b = 0.G`.

"BrHenAdd.miz" 85c ≡

```
    thus thesis by H5,GCD:33;
    end;  :: M
    thus thesis by M;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The second theorem concerns the call of `fract` in step `(5)` at the end of the algorithm, which requires the element `t2'` — the denominator of the constructed fraction — to be not zero.

"BrHenAdd.miz" 86a ≡

```
    theorem
    BH14:(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 &
     s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
     not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
     d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
     r2' = r2/d & s2' = s2/d &

     t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
```

[1] Note that theorems about greatest common divisors and divisibility are cited by `GCD:`n and not by the levels we introduced in chapter two or in appendix A.1 and A.3. The reason for this is that they are contained in a different MIZAR article: Once an article is accepted by the proof checker, one constructs a so-called MIZAR abstract containing only definitions and theorems, but no proofs. During this construction new labels for theorems (and definitions) are automatically constructed consisting of the article's name followed by a number. These levels allow one to reference these theorems in other articles.

```
     t1 <> 0.G & e ∈ Amp & e = gcd(t1,d,Amp) &
     t1' = t1/e & t2' = t2/e)
    implies t2' <> 0.G
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The proof of this theorem begins like the last one by showing that `e divides t2 = r2*s2'` and `e` $\neq$ `0.G`.[1] Please note that `t2' = (r2*(s2/gcd(r2,s2)))/e`, especially the exact form of the nominator.

"BrHenAdd.miz" 86b ≡
```
    proof
    M: now
    assume H0: r2 <> 0.G & s2 <> 0.G &
               d = gcd(r2,s2,Amp) & s2' = s2/d &
               t2 = r2*s2' & t1 <> 0.G &
               e = gcd(t1,d,Amp) & t2' = t2/e;
    H1: e divides d & d divides r2 by H0,GCD:def 12;
    H4: e divides r2 by H1,GCD:2;
    H5: e divides r2*s2' by H4,GCD:7;
    H2: e <> 0.G by H0,GCD:33;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The next step consists of showing that the numerator `r2*s2'` of `t2'` does not equal zero. For that we use theorem `GCD:8` — stating that `a/b = 0.G` if and only if `a = 0.G` — to conclude `s2'` $\neq$ `0.G`:

"BrHenAdd.miz" 87a ≡
```
    H7: d <> 0.G by H0,GCD:33;
    H9: gcd(r2,s2,Amp) divides s2 by GCD:def 12;
    H8: s2/gcd(r2,s2,Amp) <> 0.G by H0,H7,H9,GCD:8;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Thus we get the desired `r2*s2'` $\neq$ `0.G` by applying the defining property of integral domains `I x*y = 0.I` $\longrightarrow$ `(x = 0.I` $\vee$ `y = 0.I)`, stated in theorem `VECTSP_2:15`. Using again theorem `GCD:8` completes the proof.

"BrHenAdd.miz" 87b ≡
```
    H6: r2*s2' <> 0.G by H0,H8,VECTSP_2:15;
    thus thesis by H6,H5,H2,GCD:8;
    end;  :: M
    thus thesis by M;
    end;
```
◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

[1] We could have conclude this at once by the theorem above. But then we would have to state at level `H0` all the assumptions necessary to apply this theorem, rather than only the ones we need to establish the current assertions.

In the following we prove two theorems directly connected with the output `t` of the algorithm, thus showing that the output `t` of the algorithm indeed fulfills the output specification. These theorems fall into two groups. One group requires the application of the theorem of Brown and Henrici of section 2.3 (referenced as `GCD:40`). The other group consists of theorems due to special cases that can therefore proved without this theorem.

We continue with a theorem of the second kind that is due to step `(3)` of the algorithm.

"BrHenAdd.miz" 88a ≡

```
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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 r2 = 1.G & s2 = 1.G & t = fract(r1+s1,1.G))
implies (t ~ r+s & t is_normalized_wrt Amp)
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

After stating the necessary assumptions we first prove that `t` is a normalized fraction. To do so, according to the definition of `is_normalized_wrt` (`Def73`) we have to show `gcd(num(t),denom(t),Amp) = 1.G` and `denom(t) ∈ Amp`. This is no trouble, because we have `denom(t) = 1.G` by assumption.

"BrHenAdd.miz" 88b ≡

```
proof
M: now assume
H0: r is_normalized_wrt Amp & s is_normalized_wrt Amp &
    r1 = num(r) & r2 = denom(r) & s1 = num(s) & s2 = denom(s) &
    r2  = 1.G & s2 = 1.G & t = fract(r1+s1,1.G);
H2: 0.G <> 1.G by VECTSP_1:31;
H1: num(t) = r1+s1 & denom(t) = 1.G by H0,H2,F1;
H3: gcd(r1+s1,1.G,Amp) = 1.G & denom(t) ∈ Amp by H1,GCD:21,GCD:32;
H5: t is_normalized_wrt Amp by H3,H1,Def73;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

To show `t ~ r+s`, we only have to take `num(r+s) = r1*s2+s1*r2` and `denom(r+s) = r2*s2`[1] and substitute `r2 = 1.G` and `s2 = 1.G` respectively to get the desired equation `num(t)*denom(r+s) = num(r+s)*denom(t)`.

"BrHenAdd.miz" 88c ≡

```
H7: num(r+s) = r1*s2+s1*r2    by H0,F2
             .= r1*1.G+s1*r2  by H0
             .= r1*1.G+s1*1.G by H0
             .= r1+s1*1.G     by VECTSP_2:1
             .= r1+s1         by VECTSP_2:1;
```

[1] Compare theorem `F2` on page 65.

```
    H8: denom(r+s) = r2*s2   by H0,F2
                   .= 1.G*s2  by H0
                   .= s2      by VECTSP_2:1
                   .= 1.G     by H0;
    H9: num(t)*denom(r+s) = (r1+s1)*denom(r+s)  by H0,H2,F1
                          .= (r1+s1)*1.G        by H8
                          .= num(r+s)*1.G       by H7
                          .= num(r+s)*denom(t)  by H0,H2,F1;
    H10: t ~ (r+s) by H9,Def76;
    thus thesis by H10,H5;
    end;  :: M
    thus thesis by M;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We conclude this section with proving a theorem that actually requires the use of the Brown/Henrici theorem. It shows that `t` computed in step (5) of the algorithm, where `d = gcd(r2,s2) = 1`, indeed is normalized and equivalent to `r+s`.

"BrHenAdd.miz" 89 ≡

```
    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 &
     s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
     not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
     d ∈ Amp & 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)
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

The first step of the proof is to establish `gcd(r1,r2,Amp) = 1.G` and `gcd(s1,s2,Amp) = 1.G`, which follows from the definition of `is_normalized_wrt`.

"BrHenAdd.miz" 90a ≡

```
    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;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

To show `gcd(num(t),denom(t),Amp) = 1.G` we apply the theorem of Brown and Henrici. This is done by extending the term `gcd(r1*s2+r2*s1,r2*s2,Amp)` — which

in fact is `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`.

```
"BrHenAdd.miz" 90b ≡
    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;
◇
```

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

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`.

```
"BrHenAdd.miz" 90c ≡
    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;
◇
```

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

It remains to show that `t ~ r+s`. This is done by the same technique we used in the preceding proof:

```
"BrHenAdd.miz" 91 ≡
    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;
◇
```

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We end by mentioning that the MIZAR article `BrHenAdd.miz` we just presented indeed serves as a correctness proof for the generic Brown/Henrici addition algorithm, because the theorems proved in this article (by machine assistance) enable the construction of a Hoare calculus derivation for this algorithm.

Chapter 5

Verification of a Generic Euclidean Algorithm

In this section we will prove MIZAR theorems that enable the construction of a Hoare calculus derivation for the generic Euclidean algorithm of chapter 1.2. Unfortunately so far the MIZAR library does not contain Euclidean domains. Therefore we introduce this algebraic structure in section 5.1. Subsequently we give MIZAR proofs for the verification conditions constructed by our verification condition generator.[1]

5.1 Definition of Euclidean Domains

In the following we define Euclidean domains in MIZAR as well as their corresponding degree functions. We include this definitions together with the verification proofs concerning the generic Euclidean algorithm in an extra MIZAR article. Consequently, we have start with the necessary environment. For completion we also give the file `EUCL.VOC` which contains the vocabulary items introduced in the text proper.

"eucl.voc" 92a ≡

```
VEuclidean
MEuclideanRing
MDegreeFunction
```

◇

"eucl.miz" 92b ≡

```
environ
  vocabulary
  VECTSP_1,VECTSP_2,REAL_1,LINALG_1,FUNC,GCD,EUCL;
  notation
  TARSKI,ARYTM,STRUCT_0,RLVECT_1,VECTSP_1,VECTSP_2,FUNCT_2,
  NAT_1,PRELAMB,GCD;
  constructors
  NAT_1,ALGSTR_1,VECTSP_1,VECTSP_2,ARYTM,PRELAMB,GCD;
  definitions
  TARSKI,GCD;
  theorems
```

[1] Compare section 3.2.

```
  TARSKI,VECTSP_1,VECTSP_2,GCD;
  clusters
  STRUCT_0,VECTSP_1,VECTSP_2,FUNCT_2,GCD;
  schemes
  NAT_1;
  requirements
  ARYTM;

begin

reserve I for domRing;
reserve a,b,c for Element of the carrier of I;
⟨lemma for Euclidean algorithm 133⟩
```
◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

To introduce Euclidean domains, we use the already defined integral domains (or `domRing` as it is called in MIZAR). An Euclidean domain is an integral domain fulfilling the following attribute.

"eucl.miz" 93 ≡
```
definition
let I be domRing;
attr I is Euclidean means :Def1:
 ex f being Function of the carrier of I,NAT st
 (for a,b being Element of the carrier of I st b <> 0.I holds
 (ex q,r being Element of the carrier of I st
 (a = q*b+r & (r = 0.I or f.r < f.b))));
end;
```
◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

Each MIZAR type must have nonempty denotation.[1] As a consequence before defining mode `EuclideanRing` as `Euclidean domRing`, we have to show the existence of such a domain. Formally this is done with an *existential cluster*:

"eucl.miz" 94 ≡
```
definition
 cluster Euclidean domRing;
existence
proof
 ⟨existence proof for Euclidean domains 95a, ... ⟩
end;

definition
 mode EuclideanRing is Euclidean domRing;
end;
```
◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

To show existence of `Euclidean domRing` we have to construct a MIZAR object fulfilling all the requirements of the definition of `Euclidean` and `domRing`. Fortunately

[1] This prohibits types like `empty non-empty set`.

we do not have to do this from scratch, but can use mode `Field` and show that every function f is an Euclidean function for fields[1] by using the fact that in a field each element `b` $\neq$ `0` has a multiplicative inverse:

⟨existence proof for Euclidean domains 95a⟩ ≡

```
L:for F being Field
for f being Function of the carrier of F, NAT holds
(for a,b being Element of the carrier of F st b <> 0.F holds
(ex q,r being Element of the carrier of F st
(a = q*b+r & (r = 0.F or f.r < f.b))))
proof
let F be Field;
let f be Function of the carrier of F, NAT;
H2: now let a,b be Element of the carrier of F;
assume H3: b <> 0.F;
consider x being Element of the carrier of F such that
H5: b*x = 1.F by H3,VECTSP_1:30;
H6:  (a*x)*b+0.F
   = a*(b*x)+0.F  by VECTSP_1:28
  .= a*1.F+0.F    by H5
  .= a+0.F        by VECTSP_1:29
  .= a            by VECTSP_1:25;
thus b <> 0.F implies
     (ex q,r being Element of the carrier of F st
     (a = q*b+r & (r = 0.F or f.r < f.b)))  by H6;
end;  :: H2
thus thesis by H2;
end;
```

◇
Definition defined by parts 95ab.
Definition referenced in part 94.

The rest of the existence proof is easy: A field `F` is an integral domain simply by the MIZAR theorem `VECTSP_2:13`, hence using the just proven lemma `L` we can complete the proof.[2]

⟨existence proof for Euclidean domains 95b⟩ ≡

```
consider F being Field;
reconsider F as domRing by VECTSP_2:13;
consider f being Function of the carrier of F,NAT;
H2: (for a,b being Element of the carrier of F st b <> 0.F holds
    (ex q,r being Element of the carrier of F st
     (a =q*b+r & (r = 0.F or f.r < f.b)))) by L;
H3: F is Euclidean by H2,Def1;
thus thesis by H3;
end;
```

◇
Definition defined by parts 95ab.
Definition referenced in part 94.

In addition we define the mode `DegreeFunction` of an Euclidean domain. Existence of degree functions trivially follows from the definition above:

[1] In fact this proves that every field is an Euclidean domain.

[2] Note that we have to change the type of `F` using `reconsider`. The reason is that the attribute `Euclidean` is defined for integral domains only — and not for fields.

"eucl.miz" 96a ≡
```
    definition
    let E be EuclideanRing;
    mode DegreeFunction of E ->
                    Function of the carrier of E, NAT means :Def2:
     (for a,b being Element of the carrier of E st b <> 0.E holds
     (ex q,r being Element of the carrier of E st
     (a = q*b+r & (r = 0.E or it.r < it.b))));
    existence by Def1;
    end;
```
◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

A second problem concerns the existence of a greatest common divisor function in Euclidean domains.[1] We defined the greatest common divisor according to the paradigm of genericity in the most general way: for gcd domains.[2] Clearly this implies the existence of a greatest common divisor function in every gcd domain — especially in Euclidean domains. But MIZAR does not know that Euclidean domains are gcd domains. So we first have to prove this.[3]

"eucl.miz" 96b ≡

⟨Euclidean domain is gcd domain 103a, ... ⟩

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

To include the just mentioned property into the MIZAR type hierarchy (which frees us from referencing this theorem each time we talk about a greatest common divisor function in Euclidean domains), we use the *conditional cluster*:

"eucl.miz" 96c ≡
```
    definition
     cluster Euclidean -> gcd-like domRing;
    coherence by EG;
    end;
```
◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

Now we have collected all we need to prove the Euclidean algorithm of section 1.2 correct using the MIZAR system. Note that we extended the MIZAR type system by Euclidean domains, thus also gave a proof for a new SUCHTHAT global declaration, namely for `let Euclidean domain be gcd domain`.

5.2 Proofs of the Verification Conditions

In this section we prove the verification conditions for the Euclidean algorithm we already presented at the end of section 3.2. As usual we start with translating the global and local declarations of the algorithm into the MIZAR language.

[1] Do not confound this with algorithms computing the greatest common divisor function: Such a function exists in every gcd domain by just attaching to each pair of elements the corresponding greatest common divisor, whereas algorithms for this function may be varying or even non existent.

[2] Compare section 4.3 and section 2.3.

[3] This theorem will be proved in section 5.2.

"eucl.miz" 97a ≡

```
    reserve E for EuclideanRing;
    reserve d for DegreeFunction of E;
    reserve Amp for AmpleSet of E;
    reserve a,b,c for Element of the carrier of E;
    reserve u,v,s,t for Element of the carrier of E;
```

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

For completeness we also state the following theorem being trivial for the MIZAR proof checker. The reason is that although the theorem trivially holds our verification condition generator did not detect this.

"eucl.miz" 97b ≡

```
    theorem
    (u = a & v = b) implies (gcd(u,v,Amp) = gcd(a,b,Amp));
```

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

The next theorem is due to step (2) of the algorithm: It captures the case that the first input variable a is zero.

"eucl.miz" 97c ≡

```
    theorem
    (u = a & v = b & u = 0.E & c ∈ Amp & c is_associated_to v)
    implies (c ∈ Amp & c = gcd(a,b,Amp))
    proof
```

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

The proof consists of two steps. First we show that the normal form of v — the local variable holding the second input variable b — equals the greatest common divisor of a and b.

"eucl.miz" 98a ≡

```
    assume H1: u = a & v = b & u = 0.E & c ∈ Amp & c is_associated_to v;
    H2: gcd(a,b,Amp) = gcd(u,v,Amp)    by H1
                     .= gcd(0.E,v,Amp)  by H1
                     .= NF(v,Amp)       by GCD:30;
```

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

The second step is to prove that c equals the normal form of v. This is an immediate consequence of the assumptions c ∈ Amp and c is_associated_to v:

"eucl.miz" 98b ≡

```
    H4: c = NF(v,Amp) by H1,GCD:def 10;
    thus thesis by H4,H2,H1;
    end;
```

◇
File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

The following theorem concerns the last step of the algorithm: It states that the normal form c computed after the *while*-loop indeed is the greatest common divisor of the input variables a and b.

"eucl.miz" 98c ≡

```
    theorem
    (gcd(u,v,Amp) = gcd(a,b,Amp) & v = 0.E &
     c ∈ Amp & c is_associated_to u)
    implies (c ∈ Amp & c = gcd(a,b,Amp))
```

◇

File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

This theorem is nearly the same as the one before. Consequently its proof is hardly more than a copy of the one above:

"eucl.miz" 98d ≡

```
    proof
    assume H1: gcd(u,v,Amp) = gcd(a,b,Amp) & v = 0.E &
               c ∈ Amp & c is_associated_to u;
    H2: gcd(a,b,Amp) = gcd(u,v,Amp)    by H1
                     .= gcd(u,0.E,Amp)  by H1
                     .= NF(u,Amp)       by GCD:30
                     .= c               by H1,GCD:def 10;
    thus thesis by H1,H2;
    end;
```

◇

File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

The last theorem we have to prove, shows correctness of the *while*-loop. In fact it proves Euclid's equation about greatest common divisors $gcd(a,b) = gcd(b, a \bmod b)$ for $b \neq 0$.[1]

"eucl.miz" 99a ≡

```
    theorem
    (gcd(u,v,Amp) = gcd(a,b,Amp) & v <> 0.E &
     u = s*v+t & (t = 0.E or d.t < d.v))
    implies (gcd(v,t,Amp) = gcd(a,b,Amp))
```

◇

File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

To prove the theorem we first show that `gcd(v,t,Amp)` and `gcd(a,b,Amp)` are associates of each other:

"eucl.miz" 99b ≡

```
    proof
    assume H1: gcd(u,v,Amp) = gcd(a,b,Amp) & v <> 0.E &
               u = s*v+t & (t = 0.E or d.t < d.v);
    H2: gcd(v,t,Amp) divides gcd(u,v,Amp)
        ⟨proof of H2 100a⟩
    H3: gcd(u,v,Amp) divides gcd(v,t,Amp)
        ⟨proof of H3 100b⟩
    H4: gcd(u,v,Amp) is_associated_to gcd(v,t,Amp)
        by H2,H3,GCD:def 3;
```

◇

File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

By the definition of the greatest common divisor function both `gcd(v,t,Amp)` and `gcd(u,v,Amp)` are a member of the ample set `Amp`. Thus they are equal, and we can complete the proof using the assumption `gcd(u,v,Amp) = gcd(a,b,Amp)`.

[1] Compare the description of the algorithm on page 4.

"eucl.miz" 99c ≡

```
    H5: gcd(u,v,Amp) is Element of Amp by GCD:def 12;
    H6: gcd(v,t,Amp) is Element of Amp by GCD:def 12;
    H7: gcd(v,t,Amp) = gcd(u,v,Amp)  by H4,H5,H6,GCD:22
                    .= gcd(a,b,Amp)  by H1;
    thus thesis by H7;
    end;
```

◇

File defined by parts 92b, 93, 94, 96abc, 97abc, 98abcd, 99abc.

We end with the proofs of the division properties of `gcd(v,t,Amp)` and `gcd(a,b,Amp)` necessary to complete the above proof. They are easy done requiring only fundamental facts of the predicate `divides` and the greatest common divisor function.

⟨proof of H2 100a⟩ ≡

```
    proof
    M1: gcd(v,t,Amp) divides t & gcd(v,t,Amp) divides v by GCD:27;
    M3: v divides s*v by GCD:6;
    M4: gcd(v,t,Amp) divides s*v by M1,M3,GCD:2;
    M5: gcd(v,t,Amp) divides s*v+t by M4,M1,L1;
    M6: gcd(v,t,Amp) divides u by M5,H1;
    thus thesis by M1,M6,GCD:def 12;
    end;
```

◇

Definition referenced in part 99b.

⟨proof of H3 100b⟩ ≡

```
    proof
    M1: gcd(u,v,Amp) divides u & gcd(u,v,Amp) divides v by GCD:27;
    M3: v divides s*v by GCD:6;
    M4: gcd(u,v,Amp) divides s*v by M1,M3,GCD:2;
    M5: gcd(u,v,Amp) divides u-(s*v) by M1,M4,L1;
    M6: t = u-(s*v) by H1,VECTSP_2:22;
    M7: gcd(u,v,Amp) divides t by M5,M6;
    thus thesis by M1,M7,GCD:def 12;
    end;
```

◇

Definition referenced in part 99b.

Chapter 6

MIZAR and Algebraic Typechecking

Now that we have seen how to prove that a generic algebraic algorithm fulfills its specification using MIZAR, we come back to the second kind of verification concerning generic algorithms we mentioned in section 1.4: the problem of instantiations. We pointed out that it is by no means trivial in the field of computer algebra to show that a particular domain fulfills the requirements given by the algorithm. This question is deeply connected with SUCHTHAT's global declarations. In the following we discuss how to treat this problem using the MIZAR system.

6.1 Global Declarations in SUCHTHAT

Global declarations enable a SUCHTHAT user to build the algebraic environment necessary for a generic algebraic algorithm: They introduce the algebraic objects the algorithm shall deal with. Furthermore, based on these global declarations the SUCHTHAT type checker tests whether a particular instantiation is correct with respect to a generic algorithm.

SUCHTHAT declarations fall into two categories. The first one — which we already used in our example algorithms — allows one to introduce identifiers for algebraic objects, for example

```
let R be Ring,
let G be gcdDomain,
let Amp be AmpleSet of G.
```

The other kind of SUCHTHAT declarations refers to mathematical theorems, thus relating two mathematical structures. Examples are

```
let GF(p) be prime field,
let prime field be field,
let GF(p) also be vector space.
```

All these declarations are loaded in a so-called algebraic database used by the type checker. Following the implications given by the declarations, the type checker decides whether the actual parameters of a procedure call fulfill the necessary requirements

given by the specification of the formal parameters. Consequently the amount and the kind of declarations is crucial for the acceptance of an instantiated generic algebraic algorithm by the typechecker.[1] For example, the above declarations state that an algorithm written for arbitrary fields is correctly instantiated if called with `GF(p)` — provided that `p` is a prime. The Brown/Henrici addition algorithm called with the integers will be accepted if the algebraic database allows inferring that the integers are a gcd domain.[2]

Note that there is no check whether the declarations represent valid mathematical theorems. For instance, no error message will occur if a user declares the following.

```
let ring be field
```

Consequently, algorithms over fields will be considered correct if called with the integers. This obviously leads to runtime errors due to the inversion operation of fields being non existent for integers. So it seems natural to look for possibilities to verify mathematical correctness of SuchThat declarations.

For already mentioned reasons the Mizar system is capable of expressing and to proving such theorems. We will give an extended example in the next section. Note that we do not propose to run Mizar to check global declarations at compile time. This would lead to an unacceptable loss of efficiency. Instead we suggest verifying the theorems contained in the algebraic database,[3] thus improving the reliability of the knowledge the type checker uses.

Furthermore there are cases where such type questions reach into the verification of the generic algorithms themselves. Consider as an example again our Euclidean algorithm: We defined the greatest common divisor function for arbitrary gcd domains. Consequently, to use this function in Euclidean domains we must tell the Mizar proof checker that Euclidean domains are a special kind of gcd domains. But this implies that generic algorithms written for arbitrary gcd domains are correctly instantiated by Euclidean domains; in other words the correctness of the following global SuchThat declaration

```
let EuclideanRing be gcdDomain.
```

The next section gives an example for a Mizar proof of theorems arising during the verification of SuchThat global declarations, namely for the property of Euclidean domains we just presented.

6.2 Proving Declarations Correct

In this section we want to give an example for proving global SuchThat declarations correct using Mizar — not at least to present once again the fascinating facilities of the Mizar proof script language. We will show the following theorem:

[1] We believe that in a future version of SuchThat the algebraic database will contain some built-in knowledge concerning important algebraic domains.

[2] Properties of integers are an example for such built-in knowledge.

[3] This is the same approach used before, when showing correctness of generic algebraic algorithms with respect to their specifications: Obviously the proofs are not done during compilation; they are rather to raise certainty of the algorithmic library.

⟨Euclidean domain is gcd domain 103a⟩ ≡

```
theorem
EG:for E being EuclideanRing holds E is gcdDomain
```

◇
Definition defined by parts 103ab.
Definition referenced in part 96b.

From the mathematical point of view this is just a theorem connecting two algebraic domains. However from the generic algorithmic point of view it states that arbitrary algorithms written for gcd domains can be correctly instantiated by Euclidean domains; in particular the generic Brown/Henrici addition algorithm is correct (for Euclidean domains), if greatest common divisors are computed by the Euclidean algorithm of section 1.2.

To prove this theorem we have to show that Euclidean domains fulfill the predicate `gcd-like`:[1]

⟨Euclidean domain is gcd domain 103b⟩ ≡

```
proof
let E be EuclideanRing;
M: now let d be DegreeFunction of E;
N: E is gcd-like
    ⟨proof of N 104a⟩
thus thesis by N;
end;  :: M
thus thesis by M;
end;
```

◇
Definition defined by parts 103ab.
Definition referenced in part 96b.

To be more precise, we have to prove that for arbitrary elements `x` and `y` of `E` there exists a greatest common divisor `z`. We proceed by case distinction.

⟨proof of N 104a⟩ ≡

```
proof
let x,y be Element of the carrier of E;
M1: now per cases;
case A: x = 0.E;
        ⟨proof of case A 104b⟩
case B: x <> 0.E;
        ⟨proof of case B 105a, ... ⟩
end;  :: cases
thus thesis by M1;
end;
```

◇
Definition referenced in part 103b.

The first case — `x = 0.E` — is rather trivial. We simply show that `y` is a greatest common divisor of `y` and `0.E`:[2]

[1] Compare the definition of `gcd-like` on page 57.

[2] Note that showing `E` to be a gcd domain does not involve ample sets. It suffices to show that a greatest common divisor exists. Ample sets are necessary only for defining a greatest common divisor function to get a unique result.

⟨proof of case A 104b⟩ ≡

```
A1: y divides y by GCD:2;
A2: y*0.E = 0.E by VECTSP_2:26;
A3: y divides 0.E by A2,GCD:def 1;
A4: y divides x by A3,A;
A5: for zz being Element of the carrier of E
    st (zz divides x & zz divides y)
    holds (zz divides y);
thus thesis by A1,A5,A4;
```

◇

Definition referenced in part 104a.

The other case requires some more work. We follow the ideal theoretic proof given in [Lip81].[1] We start by setting `M` to the set of linear combinations of `x` and `y` (the ideal generated by `x` and `y`):

⟨proof of case B 105a⟩ ≡

```
set M = { z where z is Element of the carrier of E:
          ex s,t being Element of the carrier of E
          st z = s*x+t*y};
B1: x ∈ M & y ∈ M
    ⟨proof of B1 111b⟩
```

◇

Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

The key to the proof is to take an element $\mathtt{g} \neq \mathtt{0.E}$ with minimal degree `d.g` out of `M`.[2] To do so, we first have to show, that such an element `g` exists. We use the following MIZAR scheme stating that every subset of the natural numbers contains a minimal element:

```
scheme Min { P[Nat] } :
  ex k st P[k] & for n st P[n] holds k ≤ n
  provided ex k st P[k];
```

So we define `P[Nat]` suitable for our situation — `P` holds for the natural number `n`, if there is an element $0.E \neq \mathtt{z} \in \mathtt{M}$ having `n` as its degree — and prove the necessary precondition about `P` to apply scheme `Min`.

⟨proof of case B 105b⟩ ≡

```
defpred P[Nat] means
   ex z being Element of the carrier of E
   st (z ∈ M & z <> 0.E & 1 = d.z);
B2: ex k being Nat st P[k]
    proof
    B21: x ∈ M & x <> 0.E by B,B1;
    B23: ex k being Nat st k = d.x;
    thus thesis by B21,B23;
    end;
```

[1] Note that using the same method one can show that Euclidean domains are principal ideal domains.

[2] The element `g` will turn out to be a greatest common divisor of `x` and `y`.

```
    consider k being Nat such that
    B4: P[k] & for n being Nat st P[n] holds k ≤ n from Min(B2);
```
◇
Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

Now we can choose the desired element g being the element z of E for which P[k] holds. In addition we define G to be the set of products with g (again this actually is the ideal generated by g).

⟨proof of case B 106a⟩ ≡
```
    consider g being Element of the carrier of E such that
    B5: g ∈ M & g <> 0.E & k = d.g &
        for n being Nat st
        (ex z being Element of the carrier of E
        st (z ∈ M & z <> 0.E & n = d.z)) holds k ≤ n  by B4;
    set G = { z where z is Element of the carrier of E:
              ex r being Element of the carrier of E st z = r*g};
```
◇
Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

In fact the main effort to show that the element g is a greatest common divisor of x and y goes into proving the following identity of M and G. But before we give the proof, we want to show how to conclude using this property that g is a greatest common divisor of x and y.

⟨proof of case B 106b⟩ ≡
```
    B11: M = G
```
        ⟨proof of B11 109a⟩
◇
Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

The above equation allows us to infer x ∈ G and y ∈ G, from which we immediately conclude the first two requirements of a greatest common divisor for x and y.

⟨proof of case B 106c⟩ ≡
```
    B12: g divides x & g divides y
    proof
    H1: x ∈ G & y ∈ G by B11,B1;
    consider zx being Element of the carrier of E such that
    H2: x = zx &
        ex r being Element of the carrier of E st zx = r*g by H1;
    consider zy being Element of the carrier of E such that
    H3: y = zy &
        ex r being Element of the carrier of E st zy = r*g by H1;
    thus thesis by H2,H3,GCD:def 1;
    end;
```
◇
Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

It remains to show the "greatest property" of g. This is done by decomposing x, y and g and some equational reasoning. First we decompose x and y into products with z due to the assumptions x divides z and y divides z.

⟨proof of case B 108a⟩ ≡

```
B13: for z being Element of the carrier of E
     holds (z divides x & z divides y) implies z divides g
     proof
     let z be Element of the carrier of E;
     assume H1: z divides x & z divides y;
     consider u being Element of the carrier of E such that
     H2: x = z*u by H1,GCD:def 1;
     consider v being Element of the carrier of E such that
     H3: y = z*v by H1,GCD:def 1;
```

◇

Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

On the other hand g ∈ M = $\{z \mid \exists s,t : z = s * x + t * y\}$ gives us the following decomposition of g into a sum of products with x and y.

⟨proof of case B 108b⟩ ≡

```
consider zz being Element of the carrier of E such that
H4: g = zz &
    ex s,t being Element of the carrier of E st
    zz = s*x+t*y by B5;
consider s,t being Element of the carrier of E such that
H5: zz = s*x+t*y by H4;
```

◇

Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

Substituting the products for x and y in the decomposition of g (and some equational reasoning based on the arithmetics of E) shows, that g is a product of z, hence the desired g divides z completing the proof of our theorem.

⟨proof of case B 108c⟩ ≡

```
H6: g = s*x+t*y          by H4,H5
     .= s*(u*z)+t*y      by H2
     .= s*(u*z)+t*(v*z)  by H3
     .= (s*u)*z+t*(v*z)  by VECTSP_1:def 16
     .= (s*u)*z+(t*v)*z  by VECTSP_1:def 16
     .= (s*u+t*v)*z      by VECTSP_2:1;
thus thesis by H6,GCD:def 1;
end;
thus thesis by B12,B13;
```

◇

Definition defined by parts 105ab, 106abc, 108abc.
Definition referenced in part 104a.

In the rest of this section we present the proof of M = G to fill in the gap we left in the proof above. As usual we show such a statement by two implications.

⟨proof of B11 109a⟩ ≡

```
proof
B6: for z being Any holds z ∈ M implies z ∈ G
    ⟨proof of B6 109b, ... ⟩
B7: for z being Any holds z ∈ G implies z ∈ M
    ⟨proof of B7 110a, ... ⟩
thus thesis by B6,B7,TARSKI:2;
end;
```

◇

Definition referenced in part 106b.

To prove M ⊆ G we apply the Euclidean property of E: We divide an arbitrary z ∈ M by g getting the following.[1]

⟨proof of B6 109b⟩ ≡

```
proof
let z be Any;
assume B61: z ∈ M;
consider z2 being Element of the carrier of E such that
B67: z = z2 &
     ex s,t being Element of the carrier of E st
     z2 = s*x+t*y by B61;
reconsider z as Element of the carrier of E by B67;
consider q,r being Element of the carrier of E such that
B62: z = q*g+r & (r = 0.E or d.r < d.g) by B5,Def2;
```

◇

Definition defined by parts 109bc.
Definition referenced in part 109a.

After showing r ∈ M, we can conclude r = 0.E due to the minmality of g's degree. This implies z = q*g, hence the desired z ∈ G = $\{x \mid \exists y : x = y * g\}$.

⟨proof of B6 109c⟩ ≡

```
B63: r ∈ M
     ⟨proof of B63 111c⟩
B64: r = 0.E by B62,B63,B5;
B65: z = q*g+r    by B62
       .= q*g+0.E  by B64
       .= q*g      by VECTSP_2:1;
thus thesis by B65;
end;
```

◇

Definition defined by parts 109bc.
Definition referenced in part 109a.

The proof of the other direction G ⊆ M again is done by taking the decompositions of z and g followed by a suitable substitution. We start with decomposing an arbitrary z ∈ G into z = u*g.

[1] Note that we have to change the type of z from Any to Element of the carrier of E using reconsider before the operations of E are applicable to z.

⟨proof of B7 110a⟩ ≡

```
proof
let z be Any;
assume B71: z ∈ G;
consider z2 being Element of the carrier of E such that
B72: z = z2 &
     ex s being Element of the carrier of E st
     z2 = s*g by B71;
reconsider z as Element of the carrier of E by B72;
consider u,v being Element of the carrier of E such that
B73: z2 = u*g by B72;
B74: z = u*g by B72,B73;
```

◇
Definition defined by parts 110ab, 111a.
Definition referenced in part 109a.

In addition we know that `g = s*x+t*y` for suitable elements `s` and `t`, because `g` is an Element of the set `M` of linear combinations of `x` and `y`.

⟨proof of B7 110b⟩ ≡

```
consider z1 being Element of the carrier of E such that
B75: g = z1 &
     ex s,t being Element of the carrier of E st
     z1 = s*x+t*y by B5;
consider s,t being Element of the carrier of E such that
B76: z1 = s*x+t*y by B75;
B77: g = s*x+t*y by B75,B76;
```

◇
Definition defined by parts 110ab, 111a.
Definition referenced in part 109a.

Like above substituting `g` by `s*x+t*y` in `u*g` gives the desired representation of `z` in terms of products with `x` and `y`:

⟨proof of B7 111a⟩ ≡

```
B78: z = u*g              by B74
       .= u*(s*x+t*y)     by B77
       .= u*(s*x)+u*(t*y) by VECTSP_2:1
       .= (u*s)*x+u*(t*y) by VECTSP_1:def 16
       .= (u*s)*x+(u*t)*y by VECTSP_1:def 16;
thus thesis by B78;
end;
```

◇
Definition defined by parts 110ab, 111a.
Definition referenced in part 109a.

We conclude this section with giving the proofs of some technical statements we left out above. The first one states that `x` and `y` are contained in the set `M` — which was the ideal generated by `x` and `y`.

⟨proof of B1 111b⟩ ≡

```
proof
H1:   1.E*x+0.E*y
    = 1.E*x+0.E   by VECTSP_2:26
   .= 1.E*x       by VECTSP_2:1
   .= x           by VECTSP_2:1;
H2:   0.E*x+1.E*y
    = 0.E+1.E*y   by VECTSP_2:26
   .= 1.E*y       by VECTSP_2:1
   .= y           by VECTSP_2:1;
thus thesis by H1,H2;
end;
```

◇

Definition referenced in part 105a.

The second proof shows that `r` is an Element of `M` if `r` is given by `z = q*g+r`. We needed this fact to conclude `r = 0.E` in the proof of `M = G`.

⟨proof of B63 111c⟩ ≡

```
proof
H1:   z+(-(q*g))
   = (q*g+r)+(-(q*g))   by B62
  .= (r+q*g)+(-(q*g))
  .= r+((q*g)+(-(q*g))) by VECTSP_2:1
  .= r+0.E              by VECTSP_2:1
  .= r                  by VECTSP_2:1;
consider z1 being Element of the carrier of E
such that H2: g = z1 &
   ex s,t being Element of the carrier of E st
   z1 = s*x+t*y by B5;
consider s,t being Element of the carrier of E
such that H3: z1 = s*x+t*y by H2;
H4: g = s*x+t*y by H2,H3;
consider u,v being Element of the carrier of E
such that B68: z2 = u*x+v*y by B67;
B69: z = u*x+v*y by B67,B68;
H5: r = (u+((-q)*s))*x+(v+((-q)*t))*y
    ⟨proof of H5 112⟩
thus thesis by H5;
end;
```

◇

Definition referenced in part 109c.

The following last proof is part of the one we just presented. It is the equational reasoning necessary to show level `H5` stating the required linear combination of `r` in terms of `x` and `y`.

⟨proof of H5 112⟩ ≡

```
proof
H: r = z+(-(q*g)) by H1
    .= z+(-(q*(s*x+t*y))) by H4
    .= z+(-(q*(s*x)+q*(t*y))) by VECTSP_2:1
    .= z+((-(q*(s*x)))+(-(q*(t*y))))
       by VECTSP_2:25
    .= (u*x+v*y)+((-(q*(s*x)))+(-(q*(t*y))))
       by B69
    .= ((u*x+v*y)+(-(q*(s*x))))+(-(q*(t*y)))
```

```
          by VECTSP_2:1
       .= ((u*x+(-(q*(s*x))))+v*y)+(-(q*(t*y)))
          by VECTSP_2:1
       .= (u*x+(-(q*(s*x))))+(v*y+(-(q*(t*y))))
          by VECTSP_2:1
       .= (u*x+((-q)*(s*x)))+(v*y+(-(q*(t*y))))
          by VECTSP_2:28
       .= (u*x+((-q)*(s*x)))+(v*y+((-q)*(t*y)))
          by VECTSP_2:28
       .= (u*x+((-q)*s)*x)+(v*y+((-q)*(t*y)))
          by VECTSP_1:def 16
       .= (u*x+((-q)*s)*x)+(v*y+((-q)*t)*y)
          by VECTSP_1:def 16
       .= (u+((-q)*s))*x+(v*y+((-q)*t)*y)
          by VECTSP_2:1
       .= (u+((-q)*s))*x+(v+((-q)*t))*y
          by VECTSP_2:1;
   thus thesis by H;
   end;
```

◇

Definition referenced in part 111c.

Chapter 7

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 addition and of Euclid's algorithm on the appropriate algebraic level; thus our proofs are independent of any particular instantiation. We also showed, how to support algebraic typechecking with MIZAR, and hence how to check algebraic declarations used in the generic programming language SUCHTHAT.

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

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 do mention four 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 ([MG96]), 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`.

- In a next step the search tool should be extended not only to look for algebraic keywords, but also for whole theorems. This would enable the user to look for theorems similar to the ones he wants to prove. Using such theorems may shorten the verification proofs dramatically. The verification of the Brown/Henrici addition algorithm for example would have been a third shorter if we could have

used theorems about divisibility in integral domains and about greatest common divisors.

- Though the MIZAR system provides a proof script language capable of expressing algebraic structures appropriately, reasoning about these structures sometimes is a bit large-scale. 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 ([LeC86]). It seems promising to extend MIZAR by such procedural proof techniques (compare also [Har96]).

- Finally we also would like to have a translator transforming theorems constructed by the verification condition generator from the SCHEME representation into the MIZAR proof script language as well as other technical tools making it more comfortable to use the MIZAR system.

Besides the verification method presented for generic algebraic algorithms there are two other facets of our work we consider worth mentioning:

- Defining nontrivial algebraic domains and proving properties about these domains is more than an unwelcome effort necessary to prove generic algebraic algorithms correct. It also contributes to the field of *formalized mathematics*; namely to the QED-Project ([Boy94], [Mat95]), which aims to construct a computer system representing important mathematical knowledge as well as the necessary mathematical proof techniques.

- We also consider our work as a motivation for the use of *literate programming* ([Knu84]) in the field of computer algebra. We believe that it is of considerable advantage to combine development, presentation and verification of algorithms in one document leading to more transparency and confidence in the correctness of (generic algebraic) algorithms. We plan to provide the description of some more example algorithms rigorously following this approach.

Writing generic algebraic algorithms is a hard job: One has to look for abstract algebraic domains suitable for the method one wants to implement; in addition using the constructed generic algorithms with particular instantiations again raises nontrivial algebraic questions.

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

Acknowledgements

I want to thank Sybille Schupp, David Musser and Deepak Kapur for pointing my attention to the MIZAR system as well as Andrzej Trybulec for numerous useful hints concerning the MIZAR language. I am grateful to Rüdiger Loos for supervising my work and many fruitful discussions. Finally I want to mention my colleagues at the Wilhelm-Schickard-Institut in Tübingen, special thanks go to Reinhard Bündgen, Albrecht Haug, Uwe Kreppel, Gabor Simon, Sebastian Wedeniwski and Roland Weiss.

Bibliography

[AHU74] A. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[BN94] John J. Barton and Lee R. Nackman, *Scientific and Engineering C++*, Addison-Wesley, 1994.

[BW93] Thomas Becker and Volker Weispfenning, *Gröbner Bases: A Computational Approach to Commutative Algebra*, Springer-Verlag, Heidelberg, 1993.

[Boe95] E. Börger, *Annotated Bibliography on Evolving Algebras*, in E. Börger ed., *Specification and Validation Methods*, Oxford University Press, 1995.

[Boy94] R. Boyer et al. *The QED Manifesto*, in A. Bundy, *Automated Deduction — CADE-12*, Lecture Notes in Computer Science 814, 1994.

[Bri89] Preston Briggs, *NuWeb — A Simple Literate Programming Tool*, May 1989, `preston@cd.rice.edu`.

[Bro68] Wiliam Brown, *The Complete Euclidean Algorithm*, Technical report, Bell Telephone Laboratories, June 1968.

[BCL83] Bruno Buchberger, George E. Collins and Rüdiger Loos, *Computer Algebra: Symbolic and Algebraic Computation*, Springer, Wien, 2nd edition, 1983.

[CL91] W. Clinger and J. Rees, ed., *Revised^4 Report on the Algorithmic Language Scheme*, available by anonymous ftp from `http://www-swiss.ai.mit.edu/scheme-home.html`.

[Col74] George E. Collins, *Algebraic Algorithms, chapter two: Arithmetics*, Lecture manuscript, 1974.

[CL90] George E. Collins and Rüdiger Loos, *Specification and Index of Sac-2 Algorithms*, Technical Report WSI-90-4, Wilhelm-Schickard-Institut für Informatik, 1990.

[DJ90] N. Dershowitz and J. Jouannaud, *Rewriting Systems*, in J. van Leeuwen ed., Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics, Elsevier, 1990.

[Dij76] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, New Jersey, 1976.

[Dil94] Antoni Diller, *Z — An Introduction to Formal Methods*, 2nd ed., Wiley, New York, 1994.

[ELJ94] T. Eigenschink, D. Love and A. Jaffer, *SLib — The Portable Scheme Library*, available by anonymous ftp from `http://argus.irb.hr/languages/jacal/slib_1.html`, 1994.

[Gri81] David Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

[Gur93] Yuri Gurevich, *Evolving Algebras: A tutorial Introduction*, in G. Rozenberg and A. Salmoaa, ed., *Current Trends in Theoretical Computer Science*, World Scientific, 1993.

[Har96] John Harrison, *A Mizar Mode for HOL*, in J. von Wright, J. Grundy and J. Harrison, ed., Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 1125, 1996.

[Har97] John Harrison, *Proof Style*, Technical Report 410, University of Camebridge, Computer Laboratory, January 1997.

[Hen56] Peter Henrici, *A Subroutine for Computations with Rational Numbers*, Journal of the ACM 3(1), 1956.

[Hoa69] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM(12), 1969.

[Hol94] *Introduction to the HOL Theorem Proving System*, available by anonymous ftp from `http://lal.cs.byu.edu/lal/holdoc/Description`, 1994.

[HS78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[JS92] Richard D. Jenks and Robert S. Sutor, *Axiom: The Scientific Computation System*, Springer Verlag, New York, 1992.

[KM92] Deepak Kapur and David Musser, *Tecton: a Framework for Specifying and Verifying Generic System Components*, Technical Report, SUNY, Albany and RPI, Troy, 1992.

[Kle67] Stephen C. Kleene, *Mathematical Logic*, Wiley, New York, 1967.

[Knu84] D. Knuth, *Literate Programming*, The Computer Journal 27(2), 97-111, 1984.

[LeC86] P. LeChenadec, *Canonical Forms in Finitely Presented Algebras*, Pitman, London, 1986.

[Lip71] John D. Lipson, *Chinese Remainder and Interpolation Algorithms*, in SYMSAM, pages 372-391, 1971.

[Lip81] John D. Lipson, *Elements of Algebra and Algebraic Computing*, Benjamin/Cummings Publishing Company, 1981.

[LC92] 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.

[LS96] Rüdiger Loos and Sibylle Schupp, *SuchThat User's Guide*, October 1996, `schupp@cs.rpi.edu`, 1996.

[MG96] Udi Manber and Burra Gopal, *GLIMPSE — A Tool to Search Entire File Systems*, available via anonymous ftp by `http://glimpse.cs.arizona.edu`.

[Mat95] Roman Matuszewski, ed., *The QED Workshop II*, available via anonymous ftp from `http://www.mcs.anl.gov/qed/`, October 1995

[Mus71] David Musser, *Polynomial Factorization Algorithms*, Ph.D. thesis, University of Wisconsin, 1971.

[MS96] David Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

[MS94] David R. Musser and Alexander Stepanov, *The Ada Generic Library: Linear List Processing Packages*, Springer Verlag, New York, 1994.

[Rud92] Piotr Rudnicki, *An Overview of the Mizar Project*, available by anonymous ftp from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, June 1992.

[Sch96] Sibylle Schupp, *Generic Programming — SuchThat one can build an Algebraic Library*, Ph.D. thesis, University of Tübingen, 1996.

[SL95] Sibylle Schupp and Rüdiger Loos, *Considerations for a Generic SAC Library*, Technical Report WSI-25-95, Wlihelm-Schickard-Institut für Informatik, 1995.

[Sch97a] Christoph Schwarzweller, *The Correctness of the Generic Algorithms of Brown and Henrici concerning Addition and Multiplication in Fraction Fields of gcd domains*, Journal of Formalized Mathematics, Vol.9, 1997.

[Sch97b] Christoph Schwarzweller, *Using Mizar to prove Generic Algebraic Algorithms correct*, to appear as Technical Report, Wilhem-Schickard-Institut für Informatik.

[Sim97] Martin Simons, *The Presentation of Formal Proofs*, GMD-Bericht Nr. 278, Oldenbourg, 1997.

[Str94] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1994.

[Tar38] Alfred Tarski, *Über unerreichbare Kardinalzahlen*, Fundamenta Mathematicae(30) 1938, 68-89.

[Try93] Andrzej Trybulec, *Some Features of the Mizar Language*, available by anonymous ftp from `http://web.cs.ualberta.ca:80/~piotr/Mizar`, July 1993.

[Wan96] Changqing Wang, *Integrating Tools and Methods for Rigorous Analysis of C++ Generic Library Components*, Ph.D. thesis, Department of Computer Science, Rensselaer Polytechnic Institute, 1996.

Appendix A

Additional MIZAR Code for Generic Brown/Henrici Addition

In this section we present the MIZAR code we left out in the text and additional lemmata necessary to prove the generic Brown/Henrici addition algorithm correct. Consequently the MIZAR files extracted from this document by STWEB are complete MIZAR articles that are accepted by the MIZAR proof checker.

We start with a description of the vocabulary file `GCD.VOC`:

"gcd.voc" 122 ≡

```
   Rdivides
   Ris_unit
   Ris_associated_to
   Ris_not_associated_to
   Rare_canonical_wrt
   Rcanonical
   Rare_normalized_wrt
   Ris_multiplicative
   Oadd1
   Oadd2
   Omult1
   Omult2
   Ogcd
   ONF
   OClass
   OClasses
   Vgcd-like
   MgcdDomain
   MAmpSet
   MAmpleSet
```

◇

The capitel letters preceding the entire name indicate the kind of the following symbol. For instance the letter `O` means that this symbol is (and must be) used for a function.

M denotes mode symbols R predicate symbols and V attribute symbols. The difference between predicates and attributes is that a attribute can be attached to an already existing mode to define a new one (like we did when defining mode gcdDoamin to be gcd-like domRing in section 4.3).

A.1 Lemmata about Divisibility

Let us start with lemmata about divisibility, namely about the predicates divides, is_unit, is_associated_to and the function /. Altogether we proved 23 lemmata including three further ones necessary for the verification of the generic Euclidean algorithm.

⟨text proper 123⟩ ≡

```
theorem
IDOM1:for a,b,c being Element of the carrier of I holds
a <> 0.I implies ((a*b = a*c implies b = c) &
(b*a = c*a implies b = c))
proof
let a,b,c be Element of the carrier of I;
assume H0: a <> 0.I;
K1: now assume H1: a*b = a*c;
    H2: 0.I = a*b+(-(a*b))  by VECTSP_2:1
            .= a*b+(-(a*c))  by H1
            .= a*b+a*(-c)    by VECTSP_2:28
            .= a*(b+(-c))    by VECTSP_2:1
            .= a*(b - c)     by VECTSP_1:12;
    H3: b - c = 0.I   by H2,H0,VECTSP_2:15;
    H4: c = 0.I+c       by VECTSP_2:1
          .= (b - c)+c   by H3
          .= (b+(-c))+c  by VECTSP_1:12
          .= b+(c+(-c))  by VECTSP_2:1
          .= b+0.I       by VECTSP_2:1
          .= b           by VECTSP_2:1;
    thus b = c by H4;
    end;
thus thesis by K1;
end;
```

◇

Definition defined by parts 16, 123.
Definition referenced in part 15a.

"gcd.miz" 124 ≡

```
theorem
L1a:for a,b,c,d being Element of the carrier of I holds
(b divides a & d divides c) implies b*d divides a*c
proof
let a,b,c,d be Element of the carrier of I;
assume H1: b divides a & d divides c;
consider x being Element of the carrier of I such that
H2: b*x = a by H1,Def1;
consider y being Element of the carrier of I such that
H3: d*y = c by H1,Def1;
H4: (b*d)*(y*x) = ((b*d)*y)*x  by VECTSP_1:def 16
```

```
               .= (b*(d*y))*x  by VECTSP_1:def 16
               .= (b*c)*x      by H3
               .= c*(b*x)      by VECTSP_1:def 16
               .= a*c          by H2;
thus thesis by H4,Def1;
end;

theorem
L2:for a,b,c being Element of the carrier of I holds
a is_associated_to a &
(a is_associated_to b implies b is_associated_to a) &
((a is_associated_to b & b is_associated_to c)
 implies a is_associated_to c)
proof
let A,B,C be Element of the carrier of I;
H1: A*1.I = A  by VECTSP_2:1;
H2: A divides A by H1,Def1;
H9: A is_associated_to A by H2,Def3;
M1: now
assume H3: A is_associated_to B;
H4: A divides B & B divides A by H3,Def3;
thus A is_associated_to B implies
     B is_associated_to A by H4,Def3;
end;  :: M1
M2: now
assume H5: A is_associated_to B & B is_associated_to C;
H6: A divides B & B divides A by H5,Def3;
H7: B divides C & C divides B by H5,Def3;
H8: A divides C & C divides A by H6,H7,L1;
thus (A is_associated_to B & B is_associated_to C)
     implies A is_associated_to C by H8,Def3;
end;  ::M2

thus thesis by H9,M1,M2;
end;

theorem
L3:for a,b,c being Element of the carrier of I holds
a divides b implies c*a divides c*b
proof
let A,B,C be Element of the carrier of I;
assume H1: A divides B;
consider D being Element of the carrier of I such that
H2: A*D = B by H1,Def1;
H3: (C*A)*D = C*(A*D)  by VECTSP_1:def 16
           .= C*B      by H2;
thus thesis by H3,Def1;
end;

theorem
L6:for a,b being Element of the carrier of I holds
a divides a*b & b divides a*b by Def1;
```

```
theorem
L6a:for a,b,c being Element of the carrier of I holds
a divides b implies a divides b*c
proof
let a,b,c be Element of the carrier of I;
assume H0: a divides b;
consider d being Element of the carrier of I such that
H1: a*d = b by H0,Def1;
H2: a*(d*c) = (a*d)*c  by VECTSP_1:def 16
          .= b*c      by H1;
H3: a divides b*c by H2,Def1;
thus thesis by H3;
end;

theorem
for a,b being Element of the carrier of I holds
(b divides a &  b <> 0.I)
implies (a/b = 0.I iff a = 0.I)
proof
let a,b be Element of the carrier of I;
assume H0: b divides a & b <> 0.I;
K1: now assume H1: a/b = 0.I;
consider d being Element of the carrier of I such that
H2: d = a/b;
H2a: d = 0.I by H1,H2;
H3: a = d*b    by H2,H0,Def5
     .= 0.I*b  by H2a
     .= 0.I    by VECTSP_2:26;
thus a/b = 0.I implies a = 0.I by H3;
end;  :: K1
K2: now assume H1: a = 0.I;
consider d being Element of the carrier of I such that
H2: d = a/b;
H3: 0.I = a    by H1
       .= d*b  by H2,H0,Def5;
H4: d = 0.I by H3,H0,VECTSP_2:15;
thus a = 0.I implies a/b = 0.I by H2,H4;
end;  :: K2
thus thesis by K1,K2;
end;

theorem
L7:for a being Element of the carrier of I holds
a <> 0.I implies a/a = 1.I
proof
let A be Element of the carrier of I;
assume H0: A <> 0.I;
consider A1 being Element of the carrier of I such that
H1: A1 = A/A;
H2: A divides A by L1;
H3: A1*A = A      by H0,H1,H2,Def5
        .= 1.I*A  by VECTSP_2:1;
```

```
H5: A1 = 1.I by H3,H0,IDOM1;
thus thesis by H1,H5;
end;

theorem
for a being Element of the carrier of I holds a/1.I = a
proof
let a be Element of the carrier of I;
consider A being Element of the carrier of I such that
H0: A = a/1.I;
H1: 1.I <> 0.I by VECTSP_1: def 21;
H2: 1.I*a = a by VECTSP_2:1;
H3: 1.I divides a by H2,Def1;
H4: A = A*1.I by VECTSP_2:1
     .= a     by H0,H1,H3,Def5;
thus thesis by H4,H0;
end;

theorem
L8:for a,b,c being Element of the carrier of I holds
c <> 0.I implies
(((c divides a*b & c divides a) implies (a*b)/c = (a/c)*b) &
 ((c divides a*b & c divides b) implies (a*b)/c = a*(b/c)))
proof
let A,B,C be Element of the carrier of I;
assume H0: C <> 0.I;
K1: now
assume H1: C divides A*B & C divides A;
consider A1 being Element of the carrier of I such that
H2: A1 = A*B/C;
H3: A1*C = A*B by H2,H1,H0,Def5;
consider A2 being Element of the carrier of I such that
H4: A2 = A/C;
H5: A2*C = A by H4,H1,H0,Def5;
H6: A1*C = A*B       by H3
        .= (A2*C)*B  by H5
        .= A2*(C*B)  by VECTSP_1:def 16
        .= (A2*B)*C  by VECTSP_1:def 16;
H7: A1 = A2*B by H0,H6,IDOM1;
H8: (A*B)/C = (A/C)*B by H7,H2,H4;
thus (C divides A*B & C divides A) implies
     (A*B)/C = (A/C)*B by H8;
end;  :: K1
K2: now
assume H1: C divides A*B & C divides B;
consider A1 being Element of the carrier of I such that
H2: A1 = (A*B)/C;
H3: A1*C = A*B by H2,H1,H0,Def5;
consider A2 being Element of the carrier of I such that
H4: A2 = B/C;
H5: A2*C = B by H4,H1,H0,Def5;
H6: A1*C = A*B       by H3
        .= A*(A2*C)  by H5
        .= (A*A2)*C  by VECTSP_1:def 16;
```

```
H7: A1 = A*A2 by H0,H6,IDOM1;
H8: (A*B)/C = A*(B/C) by H7,H2,H4;
thus (C divides A*B & C divides B) implies
     (A*B)/C = A*(B/C) by H8;
end;
thus thesis by K1,K2;
end;

theorem
for a,b,c being Element of the carrier of I holds
(c <> 0.I &
 c divides a & c divides b & c divides a+b)
implies (a/c)+(b/c) = (a+b)/c
proof
let a,b,c be Element of the carrier of I;
assume H0: c <> 0.I;
assume H1:  c divides a & c divides b & c divides a+b;
consider d being Element of the carrier of I such that
H2: d = a/c;
consider e being Element of the carrier of I such that
H3: e = b/c;
H4: d*c = a by H2,H1,H0,Def5;
H5: e*c = b by H3,H1,H0,Def5;
H6: a+b = d*c+e*c  by H4,H5
       .= (d+e)*c  by VECTSP_2:1;
H7: c divides c by L1;
H8: c divides (d+e)*c by H6,H1;
H9: (a+b)/c = ((d+e)*c)/c  by H6
           .= (d+e)*(c/c)  by H0,H7,H8,L8
           .= (d+e)*1.I    by H0,L7
           .= d+e          by VECTSP_2:1;
thus thesis by H9,H2,H3;
end;

theorem
for a,b,c,d being Element of the carrier of I holds
(b <> 0.I & d <> 0.I & b divides a & d divides c)
implies (a/b)*(c/d) = (a*c)/(b*d)
proof
let a,b,c,d be Element of the carrier of I;
assume H0: b <> 0.I & d <> 0.I & b divides a & d divides c;
consider x being Element of the carrier of I such that
H1: x = a/b;
consider y being Element of the carrier of I such that
H2: y = c/d;
consider z being Element of the carrier of I such that
H3: z = (a*c)/(b*d);
H4: x*b = a by H0,H1,Def5;
H5: y*d = c by H0,H2,Def5;
H6: b*d divides a*c by H0,L1a;
H7: b*d <> 0.I by H0,VECTSP_2:15;
H8: z*(b*d) = a*c          by H3,H7,H6,Def5
           .= (x*b)*(y*d)  by H4,H5
```

```
           .= x*(b*(y*d))  by VECTSP_1:def 16
           .= x*((b*y)*d)  by VECTSP_1:def 16
           .= x*(y*(b*d))  by VECTSP_1:def 16
           .= (x*y)*(b*d)  by VECTSP_1:def 16;
H9: z = x*y by H8,H7,IDOM1;
thus thesis by H9,H1,H2,H3;
end;

theorem
L9:for a,b,c being Element of the carrier of I holds
(a <> 0.I & a*b divides a*c)
implies b divides c
proof
let A,B,C be Element of the carrier of I;
assume H1: A <> 0.I & A*B divides A*C;
consider D being Element of the carrier of I such that
H2: (A*B)*D = A*C by H1,Def1;
H3: A*(B*D) = A*C by H2,VECTSP_1:def 16;
H9: (A*(B*D))/A = (A/A)*(B*D)
    proof
    M1: A divides A*(B*D) by L6;
    M2: A divides A by L1;
    thus thesis by M1,M2,H1,L8;
    end;
H10: (A*C)/A = (A/A)*C
     proof
     M1: A divides A*C by L6;
     M2: A divides A by L1;
     thus thesis by M1,M2,H1,L8;
     end;
H11: B*D = 1.I*(B*D)    by VECTSP_2:1
        .= (A/A)*(B*D)  by L7,H1
        .= (A*(B*D))/A  by H9
        .= (A*C)/A      by H3

        .= (A/A)*C      by H10
        .= 1.I*C        by L7,H1
        .= C            by VECTSP_2:1;
thus thesis by H11,Def1;
end;

theorem
L10:for a,b being Element of the carrier of I holds
(a <> 0.I & a*b = a) implies b = 1.I
proof
let A,B be Element of the carrier of I;
assume H1: A <> 0.I & A*B = A;
consider A1 being Element of the carrier of I such that
H2: A1 = A/A;
consider B1 being Element of the carrier of I such that
H3: B1 = (A*B)/A;
H6: A1 = 1.I by H2,L7,H1;
H7: (A*B)/A = (A/A)*B
    proof
```

```
    M1: A divides A*B by L6;
    thus thesis by M1,H1,L8;
    end;
  H8: B1 = (A*B)/A  by H3
        .= (A/A)*B  by H7
        .= A1*B     by H2
        .= B        by H6,VECTSP_2:1;
  H10: A1 = B1 by H1,H2,H3;
  thus thesis by H6,H10,H8;
  end;

  theorem
  L15:for a,b,c being Element of the carrier of I holds
  (c <> 0.I & c*a is_associated_to c*b)
  implies a is_associated_to b
  proof
  let A,B,C be Element of the carrier of I;
  assume H0: C <> 0.I & C*A is_associated_to C*B;
  H1: C*A divides C*B by H0,Def3;
  H2: A divides B by H1,H0,L9;
  H3: C*B divides C*A by H0,Def3;
  H4: B divides A by H3,H0,L9;
  thus thesis by H2,H4,Def3;
  end;
```

⟨example lemma 20c⟩

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

⟨lemmata for Brown/Henrici 131⟩ ≡

```
  theorem
  L1:for a,b,c being Element of the carrier of I holds
  (a divides b & a divides c) implies a divides b+c
  proof
  let a,b,c be Element of the carrier of I;
  assume H1: a divides b & a divides c;
  consider d being Element of the carrier of I such that
  H2: b = a*d by H1,GCD:def 1;
  consider e being Element of the carrier of I such that
  H3: c = a*e by H1,GCD:def 1;
  H4: a*(d+e) = a*d+a*e  by VECTSP_2:1
             .= b+c      by H2,H3;
  thus thesis by H4,GCD:def 1;
  end;

  theorem
  L3:for a,b,c being Element of the carrier of I holds
  c <> 0.I implies
  ((c divides a implies (a*b)/c = (a/c)*b) &
   (c divides b implies (a*b)/c = a*(b/c)))
  proof



  let a,b,c be Element of the carrier of I;
```

```
assume H0: c <> 0.I;
K1: now assume H1: c divides a;
M1: c divides a*b by H1,GCD:7;
consider a1 being Element of the carrier of I such that
H2: a1 = (a*b)/c;
H3: a1*c = a*b by H2,H0,M1,GCD:def 4;
consider a2 being Element of the carrier of I such that
H4: a2 = a/c;
H5: a2*c = a by H4,H1,H0,GCD:def 4;
H6: a1*c = a*b       by H3
        .= (a2*c)*b  by H5
        .= a2*(c*b)  by VECTSP_1:def 16
        .= (a2*b)*c  by VECTSP_1:def 16;
H7: a1 = a2*b by H0,H6,GCD:1;
thus c divides a implies (a*b)/c = (a/c)*b by H7,H2,H4;
end;  :: K1
K2: now assume H1: c divides b;
M1: c divides a*b by H1,GCD:7;
consider a1 being Element of the carrier of I such that
H2: a1 = (a*b)/c;
H3: a1*c = a*b by H2,H0,M1,GCD:def 4;
consider a2 being Element of the carrier of I such that
H4: a2 = b/c;
H5: a2*c = b by H4,H1,H0,GCD:def 4;
H6: a1*c = a*b       by H3
        .= a*(a2*c)  by H5
        .= (a*a2)*c  by VECTSP_1:def 16;
H7: a1 = (a*a2) by H0,H6,GCD:1;
thus c divides b implies (a*b)/c = a*(b/c) by H7,H2,H4;
end;
thus thesis by K1,K2;
end;

theorem
L2:for a,b,c being Element of the carrier of I holds
(c <> 0.I & c divides a & c divides b)
implies (a/c)+(b/c) = (a+b)/c
proof
let a,b,c be Element of the carrier of I;
assume H0: c <> 0.I & c divides a & c divides b;
consider d being Element of the carrier of I such that
H2: d = a/c;
consider e being Element of the carrier of I such that
H3: e = b/c;
H4: d*c = a by H2,H0,GCD:def 4;
H5: e*c = b by H3,H0,GCD:def 4;
H6: a+b = (d*c)+(e*c)  by H4,H5
       .= (d+e)*c      by VECTSP_2:1;
H7: c divides c by GCD:2;
H8: c divides (d+e)*c by GCD:def 1;
H9: (a+b)/c = ((d+e)*c)/c  by H6
           .= (d+e)*(c/c)  by H0,H7,H8,GCD:11

           .= (d+e)*(1.I)  by H0,GCD:9
```

```
            .= d+e          by VECTSP_2:1;
    thus thesis by H9,H2,H3;
    end;
◇
```
Definition referenced in part 164.

The remaining lemma is not necessary for the correctness proof of the generic Brown/ Henrici addition algorithm, but we used it for the one of the generic Euclidean algorithm. We put it in this section because it is also about divisibility.

⟨lemma for Euclidean algorithm 133⟩ ≡

```
    theorem
    L1:for I being domRing
    for a,b,c being Element of the carrier of I holds
    ((a divides b & a divides c) implies a divides b+c) &
    ((a divides b & a divides c) implies a divides b-c)
    proof
    let I be domRing;
    let a,b,c be Element of the carrier of I;
    M1: now assume
    H1: a divides b & a divides c;
    consider d being Element of the carrier of I such that
    H2: b = a*d by H1,GCD:def 1;
    consider e being Element of the carrier of I such that
    H3: c = a*e by H1,GCD:def 1;
    H4: a*(d+e) = a*d+a*e  by VECTSP_2:1
            .= b+c      by H2,H3;
    thus (a divides b & a divides c) implies a divides b+c
         by H4,GCD:def 1;
    end;  :: M1
    M2: now assume
    H5: a divides b & a divides c;
    consider d being Element of the carrier of I such that
    H6: b = a*d by H5,GCD:def 1;
    consider e being Element of the carrier of I such that
    H7: c = a*e by H5,GCD:def 1;
    H8: a*(d-e) = a*d-a*e  by VECTSP_2:31
            .= b-c      by H6,H7;
    thus (a divides b & a divides c) implies a divides b-c
         by H8,GCD:def 1;
    end;  :: M2
    thus thesis by M1,M2;
    end;
◇
```
Definition referenced in part 92b.

A.2 Lemmata about Ample Sets

This section contains MIZAR code for defining ample sets, multiplicative ample sets and normal forms modulo ample sets, as well as theorems proving some additional properties about these structures.

Let us start with some easy properties about the `Class` and the `Classes` function:

"gcd.miz" 134 ≡

⟨Definition of association classes 60b⟩

```
theorem
CL1:for a,b being Element of the carrier of I holds
Class a ∩ Class b <> ∅ implies Class a = Class b
proof
let a,b be Element of the carrier of I;
assume H0: Class a ∩ Class b <> ∅;
H0a: Class a meets Class b by H0,BOOLE:119;
consider Z being Any such that
H1: Z ∈ Class a & Z ∈ Class b by H0a,BOOLE:def 5;
reconsider Z as Element of the carrier of I by H1;
H4: Z is_associated_to a by H1,Defh1;
H5: Z is_associated_to b by H1,Defh1;
H6: c ∈ Class a implies c ∈ Class b
    proof
    assume H7: c ∈ Class a;
    H8: c is_associated_to a by H7,Defh1;
    H9: a is_associated_to c by H8,L2;
    H10: Z is_associated_to c by H4,H9,L2;
    H11: b is_associated_to Z by H5,L2;
    H12: b is_associated_to c by H11,H10,L2;
    H13: c is_associated_to b by H12,L2;
    H14: c ∈ Class b by H13,Defh1;
    thus thesis by H14;
    end;
H15: c ∈ Class b implies c ∈ Class a
     proof
     assume H7: c ∈ Class b;
     H16: c is_associated_to b by H7,Defh1;
     H17: b is_associated_to c by H16,L2;
     H18: Z is_associated_to c by H5,H17,L2;
     H19: a is_associated_to Z by H4,L2;
     H20: a is_associated_to c by H19,H18,L2;
     H21: c is_associated_to a by H20,L2;
     H22: c ∈ Class a by H21,Defh1;
     thus thesis by H22;
     end;
thus thesis by H6,H15,SUBSET_1:8;
end;

theorem
CL2:for I being domRing holds Classes I is non empty
proof
let I be domRing;
H1: Class (1.I) ∈ Classes I by Defh2;
thus thesis by H1;
end;

theorem
CL3:for X being Subset of the carrier of I holds
X ∈ Classes I implies X is non empty
proof
```

```
    let X be Subset of the carrier of I;
    assume H0: X ∈ Classes I;
    H1: ex a being Element of the carrier of I st X = Class a by H0,Defh2;
    thus thesis by H1;
    end;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

In the following we present the correctness proofs of the functions `Class` and `Classes`. Note that each correctness proof consists of an existence and a uniqueness proof.

⟨correctness proof of Class 135⟩ ≡
```
    existence
      proof
      set M = { b where b is Element of the carrier of I:
                b is_associated_to a};
      K1: M is non empty Subset of the carrier of I
          proof
          K2: now let B be Any;
              K3: now assume L1: B ∈ M;
              L2: ex B' being Element of the carrier of I st
                        B = B' & B' is_associated_to a by L1;
              L3: B ∈ the carrier of I by L2;
              thus B ∈ M implies B ∈ the carrier of I by L3;
              end;
              thus B ∈ M implies B ∈ the carrier of I by K3;
              end;
          L4: M c= the carrier of I by K2,TARSKI:def 3;
          L5: M is non empty
              proof
              H1: a is_associated_to a by L2;
              H2: a ∈ M by H1;
              thus thesis by H2;
              end;
          thus thesis by L4,L5;
          end;
      K4: now let A be Element of the carrier of I;
          H1: A ∈ M implies A is_associated_to a
              proof
              assume M1: A ∈ M;
              M2: ex A' being Element of the carrier of I st
                        A = A' & A' is_associated_to a by M1;
              thus thesis by M2;
              end;
          thus A ∈ M iff A is_associated_to a by H1;
          end;
      thus thesis by K1,K4;
      end;
    uniqueness
      proof
      let M,N be non empty Subset of the carrier of I;
      assume H1: for A being Element of the carrier of I holds
                 A ∈ M iff A is_associated_to a;
      assume H2: for A being Element of the carrier of I holds
```

```
              A ∈ N iff A is_associated_to a;
      H3: for a being Element of the carrier of I holds
          a ∈ M iff a ∈ N
          proof
          let A be Element of the carrier of I;
          K1: now assume M1: A ∈ M;
              M2: A is_associated_to a by H1,M1;
              M3: A ∈ N by M2,H2;
              thus A ∈ M implies A ∈ N by M3;
              end;
          K2: now assume M1: A ∈ N;
              M2: A is_associated_to a by H2,M1;
              M3: A ∈ M by M2,H1;
              thus A ∈ N implies A ∈ M by M3;
             end;
          thus thesis by K1,K2;
          end;
      thus thesis by H3,SUBSET_1:8;
      end;
    end;
```

◇

Definition referenced in part 60b.

⟨correctness proof of Classes 137⟩ ≡

```
    existence from SubFamEx;
    uniqueness
      proof
      let F1,F2 be Subset-Family of the carrier of I;
      assume A: for A being Subset of the carrier of I holds
                A ∈ F1 iff
                ex a being Element of the carrier of I st A = Class a;
      assume B: for A being Subset of the carrier of I holds
                A ∈ F2 iff
                ex a being Element of the carrier of I st A = Class a;
      thus thesis from SubFamComp(A,B);
      end;
    end;
```

◇

Definition referenced in part 60b.

Now we present to some additional properties of ample sets. The first theorem summarizes the basic properties of an ample set. The other ones prove useful statements about ample sets, we use in later MIZAR proofs.

"gcd.miz" 138 ≡

⟨Definition of AmpleSet 60a, ... ⟩

```
    theorem
    AMP:for Amp being AmpleSet of I holds
    1.I ∈ Amp &
    (for a being Element of the carrier of I ex z being Element of Amp
     st z is_associated_to a) &

    (for x,y being Element of Amp holds x <> y implies
```

```
     x is_not_associated_to y)
    proof
    let Amp be AmpleSet of I;
    H0: 1.I ∈ Amp by Def8;
    H1: Amp is AmpSet of I by Def8;
    H2: (for a being Element of the carrier of I ex z being Element of Amp
         st z is_associated_to a) &
        (for x,y being Element of Amp holds x <> y implies
          x is_not_associated_to y) by H1,Def8a;
    thus thesis by H0,H2;
    end;

    theorem
    AMP1:for x,y being Element of Amp holds
    x is_associated_to y implies x = y by AMP;

    theorem
    AMP0:for Amp being AmpleSet of I holds
    0.I is Element of Amp
    proof
    let Amp be AmpleSet of I;
    consider A being Element of Amp such that
    H0: A is_associated_to 0.I by AMP;
    H1: 0.I divides A by H0,Def3;
    consider D being Element of the carrier of I such that
    H2: 0.I*D = A by H1,Def1;
    H3: A = 0.I by H2,VECTSP_2:26;
    thus thesis by H3;
    end;
```

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

We carry on with the existence proof of the mode `AmpleSet`. Remember that this is an ample set which contains the multiplicative identity of the corresponding integral domain.

⟨existence proof of AmpleSet 139⟩ ≡

```
    existence
    proof
    H0: now ⟨Defining AmpleSet 69c⟩
    H2: 1.I ∈ A'
        proof
        M1: 1.I ∈ {1.I} by ENUMSET1:4;
        thus thesis by M1,BOOLE:def 2;
        end;
    reconsider A' as non empty set by H2;
    H3: for x being Element of A' holds x = 1.I or x ∈ A
        proof
        let y be Element of A';
        M3: now per cases by BOOLE:def 2;
        case A: y ∈ {z where z is Element of A: z <> x};
        A1: ex zz being Element of A st y = zz & zz <> x by A;
        thus thesis by A1;
        case B: y ∈ {1.I};
```

```
    thus thesis by B,ENUMSET1:3;
    end;  :: cases
    thus thesis by M3;
    end;
H4: A' is non empty Subset of the carrier of I
    proof
    M1: now let x be Any;
    M2: now assume M3: x ∈ A';
    M4: x ∈ the carrier of I
        proof
        M4a: now per cases by M3,H3;
        case A: x = 1.I;
        thus thesis by A;
        case B: x ∈ A;
        thus thesis by B;
        end;  :: cases
        thus thesis by M4a;
        end;
    thus x ∈ A' implies x ∈ the carrier of I by M4;
    end;  :: M2
    thus x ∈ A' implies x ∈ the carrier of I by M2;
    end;  :: M1
    thus thesis by M1,TARSKI:def 3;
    end;
reconsider A' as non empty Subset of the carrier of I by H4;
H5: for a being Element of the carrier of I ex z being Element of A'
    st z is_associated_to a
    proof
    let a be Element of the carrier of I;
    M0: now per cases;
    case A: a is_associated_to 1.I;
            A1: 1.I is_associated_to a by A,L2;
            thus ex z being Element of A' st z is_associated_to a
                 by A1,H2;
    case B: a is_not_associated_to 1.I;
            consider z being Element of A such that
            B1: z is_associated_to a by Def8a;
            B3: z <> x
                proof
                assume M1: z = x;
                M2: z is_associated_to 1.I by M1,H1;
                M3: a is_associated_to z by B1,L2;
                M4: a is_associated_to 1.I by M3,M2,L2;
                thus thesis by M4,B;
                end;
            B4: z ∈ {zz where zz is Element of A : zz <> x}
                by B3;
            B5: z ∈ A' by B4,BOOLE:def 2;
            thus ex z being Element of A' st z is_associated_to a
                 by B1,B5;
    end;  :: cases
    thus thesis by M0;
    end;
H6: for z,y being Element of A' holds
```

```
        z <> y implies z is_not_associated_to y
        proof
        let z,y be Element of A';
        assume M0: z <> y;
        M1: now per cases;
        case A: z = 1.I & y = 1.I;
                thus thesis by A,M0;
        case B: z = 1.I & y <> 1.I;
                B1: y ∈ A by B,H3;
                B2: not(y ∈ {1.I}) by B,ENUMSET1:3;
                B4: y ∈ {zz where zz is Element of A: zz <> x}
                    by B2,BOOLE:def 2;
                B5a: ex zz being Element of A st y = zz & zz <> x by B4;
                B5: y <> x by B5a;
                B6: x is_associated_to z by B,H1;
                assume B7: z is_associated_to y;
                B8: x is_associated_to y by B6,B7,L2;
                B10: x is_not_associated_to y by Def8a,B5,B1;
                thus thesis by B10,B8;
        case C: z <> 1.I & y = 1.I;
                C1: z ∈ A by C,H3;
                C2: not(z ∈ {1.I}) by C,ENUMSET1:3;
                C4: z ∈ {zz where zz is Element of A: zz <> x}
                    by C2,BOOLE:def 2;
                C5a: ex zz being Element of A st z = zz & zz <> x by C4;
                C5: z <> x by C5a;
                C6: x is_associated_to y by C,H1;
                C6a: y is_associated_to x by C6,L2;
                assume C7: z is_associated_to y;

                C8: z is_associated_to x by C6a,C7,L2;
                C10: z is_not_associated_to x by C5,C1,Def8a;
                thus thesis by C10,C8;
        case D: z <> 1.I & y <> 1.I;
                D1: z ∈ A by D,H3;
                D2: y ∈ A by D,H3;
                thus thesis by M0,D1,D2,Def8a;
        end;  :: cases
        thus thesis by M1;
        end;
    H7: A' is AmpSet of I by H5,H6,Def8a;
    thus thesis by H2,H7;
    end;  :: H0
    thus thesis by H0;
    end;
    end;
```

◇

Definition referenced in part 69b.

Now we present some further code concerning multiplicative ample sets, namely the proof of theorem AMP5 stating that multiplicative ample sets are also closed with respect to division.

"gcd.miz" 142a ≡

⟨Definition of multiplicative AmpleSet 70a, ... ⟩

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

⟨proof of AMP5 142b⟩ ≡

```
proof
let Amp be AmpleSet of I;
assume H0: Amp is_multiplicative;
let x,y be Element of Amp;
assume H1: y divides x & y <> 0.I;
M: now per cases;
case A: x <> 0.I;
        consider d being Element of the carrier of I such that
        H2: d = x/y;
        H2a: x = y*d by H2,H1,Def5;
        consider d' being Element of Amp such that
        H3: d' is_associated_to d by AMP;
        H3a: d is_associated_to d' by H3,L2;
        consider u being Element of the carrier of I such that
        H4: u is_unit & d*u = d' by H3a,L11;
        H5: u*x = u*(y*d)  by H2a
               .= y*(d*u)  by VECTSP_1:def 16
               .= y*d'     by H4;
        H5a: y*d' ∈ Amp by H0,Def25;
        H6: u*x ∈ Amp by H5a,H5;
        H7: x is_associated_to u*x
            proof
            M1: x divides x by L1;
            M2: x divides u*x by M1,L6a;
            M3: u divides 1.I by H4,Def2;
            consider e being Element of the carrier of I such that
            M4: u*e = 1.I by M3,Def1;
            M5: (u*x)*e = e*(u*x)
                        .= (e*u)*x  by VECTSP_1:def 16
                        .= 1.I*x    by M4
                        .= x        by VECTSP_2:1;
            M6: u*x divides x by M5,Def1;
            thus thesis by M2,M6,Def3;
            end;
        H8: 1.I*x = x   by VECTSP_2:1
                  .= u*x by H7,H6,AMP1;
        H9: u = 1.I by H8,IDOM1,A;
        H10: d' = d*u   by H4
               .= d*1.I by H9
               .= d     by VECTSP_2:1;
        thus thesis by H10,H2;
case B: x = 0.I;
        consider d being Element of the carrier of I such that
        M0: d = x/y;
        M0a: x = y*d by M0,H1,Def5;
        M1: x*y = 0.I*y  by B
               .= 0.I    by VECTSP_2:26;
        M1a: x = 0.I by VECTSP_2:15,M1,H1;
        M2: d = 0.I by VECTSP_2:15,M1a,H1,M0a;
```

```
            M3: 0.I is Element of Amp by AMP0;
            thus thesis by M0,M3,M2;
    end;  :: cases
    thus thesis by M;
    end;
```
◇
Definition referenced in part 70b.

We conclude this section with some properties of the normal form modulo an ample set. We also present the correctness proof according to the mode NF we left out at the end of section 4.2.

"gcd.miz" 143 ≡
⟨Definition of Normal Form 71a⟩

```
    theorem
    NF1:for Amp being AmpleSet of I holds
    NF(0.I,Amp) = 0.I & NF(1.I,Amp) = 1.I
    proof
    let Amp be AmpleSet of I;
    H0: 1.I is_associated_to 1.I by L2;
    H1: 1.I ∈ Amp by Def8;
    H2: NF(1.I,Amp) = 1.I by H0,H1,Def20;
    H3: 0.I is_associated_to 0.I by L2;
    H4: 0.I is Element of Amp by AMP0;
    H5: NF(0.I,Amp) = 0.I by H3,H4,Def20;
    thus thesis by H2,H5;
    end;

    theorem
    for Amp being AmpleSet of I
    for a being Element of the carrier of I holds
    a ∈ Amp iff a = NF(a,Amp)
    proof
    let Amp be AmpleSet of I;
    let a be Element of the carrier of I;
    K1: now assume H0: a ∈ Amp;
    H1: a is_associated_to a by L2;
    H2: a = NF(a,Amp) by H0,H1,Def20;
    thus a ∈ Amp implies a = NF(a,Amp) by H2;
    end;  :: K1
    thus thesis by K1,Def20;
    end;
```
◇
File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

⟨correctness proof of normal form 144⟩ ≡

```
    existence
      proof
      K: now let Amp be AmpleSet of I;
      let x be Element of the carrier of I;
      consider z being Element of Amp such that
      H0: z is_associated_to x by AMP;
      thus ex zz being Element of the carrier of I st
```

```
        zz ∈ Amp & zz is_associated_to x  by H0;
    end;  :: K
    thus thesis by K;
    end;
  uniqueness
    proof
    let z1,z2 be Element of the carrier of I such that
    H0: z1 ∈ Amp & z1 is_associated_to x &
        z2 ∈ Amp & z2 is_associated_to x;
    H0a: z1 is Element of Amp &
         z2 is Element of Amp by H0;
    H1: x is_associated_to z2 by H0,L2;
    H2: z1 is_associated_to z2 by H0,H1,L2;
    H3: z1 = z2 by H0a,H2,AMP1;
    thus thesis by H3;
    end;
  end;
```

◇

Definition referenced in part 71a.

A.3 Lemmata about Gcd Domains

In this section we give the MIZAR proofs of the lemmata about the greatest common divisor function we need to establish the theorem of Brown and Henrici presented in section 2.3. The definition of gcd domains can be found in section 4.3, the one of the greatest common divisor function in section 2.3.

"gcd.miz" 145a ≡

```
⟨Definition of gcdDomain 71b, ... ⟩
reserve I for gcdDomain;
```

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

Let us begin with the correctness — that is existence and uniqueness — proof of the greatest common divisor function in arbitrary gcd domains.

⟨correctness proof of gcd function 145b⟩ ≡

```
  existence
    proof
    consider u being Element of the carrier of I such that
    H1: u divides x &
        u divides y &
       (for zz being Element of the carrier of I
        st (zz divides x & zz divides y)
        holds zz divides u) by Def7;
    consider z being Element of Amp such that
    H2: z is_associated_to u by AMP;
    H3: z divides u by H2,Def3;
    H4: z divides x by H3,H1,L1;
    H5: z divides y by H3,H1,L1;
    H6: for zz being Element of the carrier of I
        st (zz divides x & zz divides y) holds zz divides z
        proof
        let zz be Element of the carrier of I;
```

```
        assume M1: zz divides x & zz divides y;
        M2: zz divides u by M1,H1;
        M3: u divides z by H2,Def3;
        M4: zz divides z by M2,M3,L1;
        thus thesis by M4;
        end;
      thus thesis by H4,H5,H6;
      end;
    uniqueness
      proof
      K1: now let z1 be Element of the carrier of I such that
      H1: z1 ∈ Amp &
          z1 divides x &
          z1 divides y &
          (for z being Element of the carrier of I
           st (z divides x & z divides y)
           holds z divides z1);
      let z2 be Element of the carrier of I such that
      H2: z2 ∈ Amp &
          z2 divides x &
          z2 divides y &
          (for z being Element of the carrier of I
           st (z divides x & z divides y)
           holds z divides z2);
      H3: z1 is_associated_to z2
          proof
          M1: z1 divides x & z1 divides y by H1;
          M2: z1 divides z2 by M1,H2;
          M3: z2 divides x & z2 divides y by H2;
          M4: z2 divides z1 by M3,H1;
          thus thesis by M2,M4,Def3;
          end;
      thus z1 = z2 by H1,H2,H3,AMP;
      end;  :: K1
      thus thesis by K1;
      end;
    end;
◇
```

Definition referenced in part 24.

What follows next are the above mentioned lemmata about the greatest common divisor function. We proved 13 lemmata — including the five theorems we already presented in section 2.3.

"gcd.miz" 146 ≡

⟨Definition of gcd function 24⟩

```
theorem
L0:for Amp being AmpleSet of I
for a,b being Element of the carrier of I holds
gcd(a,b,Amp) divides a & gcd(a,b,Amp) divides b by Def4;

theorem
L4:for Amp being AmpleSet of I
```

```
for a,b,c being Element of the carrier of I holds
c divides gcd(a,b,Amp) implies (c divides a & c divides b)
proof
let Amp be AmpleSet of I;
let A,B,C be Element of the carrier of I;
assume H1: C divides gcd(A,B,Amp);
consider D being Element of the carrier of I such that
H2: C*D = gcd(A,B,Amp) by H1,Def1;
H3: gcd(A,B,Amp) divides A by L0;
consider E being Element of the carrier of I such that
H4: gcd(A,B,Amp)*E = A by H3,Def1;
H5: C*(D*E) = (C*D)*E         by VECTSP_1:def 16
           .= gcd(A,B,Amp)*E  by H2
           .= A               by H4;
H6: C divides A by H5,Def1;
H7: gcd(A,B,Amp) divides B by L0;
consider E being Element of the carrier of I such that
H8: gcd(A,B,Amp)*E = B by H7,Def1;
H9: C*(D*E) = (C*D)*E         by VECTSP_1:def 16
           .= gcd(A,B,Amp)*E  by H2
           .= B               by H8;
H10: C divides B by H9,Def1;
thus thesis by H6,H10;
end;

theorem
L13:for Amp being AmpleSet of I
for a,b being Element of the carrier of I holds
gcd(a,b,Amp) = gcd(b,a,Amp)
proof
let Amp be AmpleSet of I;
let A,B be Element of the carrier of I;
consider D being Element of the carrier of I such that
H1: D = gcd(A,B,Amp);
H11: D ∈ Amp by Def4,H1;
H2: D divides B & D divides A by H1,L0;
H3: for z being Element of the carrier of I
    st (z divides B & z divides A)
    holds (z divides D) by H1,Def4;
H4: D = gcd(B,A,Amp) by H11,H2,H3,Def4;
thus gcd(A,B,Amp) = gcd(B,A,Amp) by H1,H4;
end;

theorem
GCD1:for Amp being AmpleSet of I
for a being Element of the carrier of I holds
gcd(a,0.I,Amp) = NF(a,Amp) &
gcd(0.I,a,Amp) = NF(a,Amp)
proof
let Amp be AmpleSet of I;
let A be Element of the carrier of I;
H0: NF(A,Amp)is_associated_to A by Def20;
H1: NF(A,Amp) divides A by H0,Def3;
H2: NF(A,Amp)*0.I = 0.I by VECTSP_2:26;
```

```
H3: NF(A,Amp) divides 0.I by H2,Def1;
H4: for z being Element of the carrier of I
    st (z divides A & z divides 0.I)
    holds z divides NF(A,Amp)
    proof
    let z be Element of the carrier of I;
    assume M0: z divides A & z divides 0.I;
    M1: A divides NF(A,Amp) by H0,Def3;
    thus thesis by M1,M0,L1;
    end;
H5: NF(A,Amp) ∈ Amp by Def20;
H6: gcd(A,0.I,Amp) = NF(A,Amp) by H1,H3,H4,H5,Def4;
thus thesis by H6,L13;
end;

theorem
GCD0:for Amp being AmpleSet of I holds
gcd(0.I,0.I,Amp) = 0.I
proof
let Amp being AmpleSet of I;
H2: gcd(0.I,0.I,Amp) = NF(0.I,Amp) by GCD1;
H3: NF(0.I,Amp) = 0.I by NF1;
thus thesis by H2,H3;
end;

theorem
GCD2:for Amp being AmpleSet of I
for a being Element of the carrier of I holds
gcd(a,1.I,Amp) = 1.I & gcd(1.I,a,Amp) = 1.I
proof
let Amp be AmpleSet of I;
let A be Element of the carrier of I;
H0: 1.I ∈ Amp by Def8;
H1: 1.I divides 1.I by L1;
H2: 1.I*A = A by VECTSP_2:1;
H3: 1.I divides A by H2,Def1;
H4: for z being Element of the carrier of I
    st (z divides A & z divides 1.I)
    holds z divides 1.I;
H5: gcd(A,1.I,Amp) = 1.I by H0,H1,H3,H4,Def4;
thus thesis by H5,L13;
end;

theorem
L12:for Amp being AmpleSet of I
for a,b being Element of the carrier of I holds
gcd(a,b,Amp) = 0.I iff (a = 0.I & b = 0.I)
proof
let Amp be AmpleSet of I;
let A,B be Element of the carrier of I;
H0: (A = 0.I & B = 0.I) implies gcd(A,B,Amp) = 0.I

    proof
    assume H0: A = (0.I) & B = (0.I);
```

```
    H3: gcd(A,B,Amp) = NF(A,Amp) by H0,GCD1;
    H4: NF(A,Amp) = (0.I) by H0,NF1;
    thus thesis by H4,H3;
    end;
K: now assume H1: gcd(A,B,Amp) = (0.I);
   H2: (0.I) divides A & (0.I) divides B by H1,Def4;
   consider D being Element of the carrier of I such that
   H3: 0.I*D = A by H2,Def1;
   H4: A = 0.I by H3,VECTSP_2:26;
   consider E being Element of the carrier of I such that
   H5: 0.I*E = B by H2,Def1;
   H6: B = 0.I by H5,VECTSP_2:26;
   thus gcd(A,B,Amp) = 0.I implies (A = 0.I & B = 0.I)
     by H4,H6;
   end;
thus thesis by H0,K;
end;

theorem
L14:for Amp being AmpleSet of I
for a,b,c being Element of the carrier of I holds
b is_associated_to c implies
(gcd(a,b,Amp) is_associated_to gcd(a,c,Amp) &
 gcd(b,a,Amp) is_associated_to gcd(c,a,Amp))
proof
let Amp be AmpleSet of I;
let A,B,C be Element of the carrier of I;
assume H1: B is_associated_to C;
H2: B divides C by H1,Def3;
H3: gcd(A,B,Amp) divides B & gcd(A,B,Amp) divides A by L0;
H4: gcd(A,B,Amp) divides C by H2,H3,L1;
H6: gcd(A,B,Amp) divides gcd(A,C,Amp) by H4,H3,Def4;
H7: gcd(A,B,Amp) = gcd(B,A,Amp) by L13;
H8: gcd(A,C,Amp) = gcd(C,A,Amp) by L13;
H9: gcd(B,A,Amp) divides gcd(C,A,Amp) by H6,H7,H8;
H10: C divides B by H1,Def3;
H11: gcd(A,C,Amp) divides C by L0;
H12: gcd(A,C,Amp) divides B by H10,H11,L1;
H13: gcd(A,C,Amp) divides A by L0;
H14: gcd(A,C,Amp) divides gcd(A,B,Amp) by H13,H12,Def4;
H15: gcd(C,A,Amp) divides gcd(B,A,Amp) by H7,H8,H14;
H16: gcd(A,B,Amp) is_associated_to gcd(A,C,Amp) by H6,H14,Def3;
H17: gcd(B,A,Amp) is_associated_to gcd(C,A,Amp) by H9,H15,Def3;
thus thesis by H16,H17;
end;
```

⟨gcd theorems 25⟩
⟨Brown/Henrici theorem 23b⟩

◇

File defined by parts 15a, 17ab, 18ab, 19abc, 20ab, 124, 134, 138, 142a, 143, 145a, 146.

We conclude this section with the MIZAR proofs of the remaining four theorems about the greatest common divisor function that we gave in section 2.3. Note that the proofs directly correspond to the ones that were given in [Col74].

⟨proof of theorem T0 151a⟩ ≡

```
proof
let Amp be AmpleSet of I;
let A,B,C be Element of the carrier of I;
consider D being Element of the carrier of I such that
H1: D = gcd(gcd(A,B,Amp),C,Amp);
consider E being Element of the carrier of I such that
H2: E = gcd(A,gcd(B,C,Amp),Amp);
H3: D divides gcd(A,B,Amp) & D divides C by H1,L0;
H4: D divides A & D divides B & D divides C by H3,L4;
H5: D divides A & D divides gcd(B,C,Amp) by H4,Def4;
H6: D divides E by H2,H5,Def4;
H7: E divides gcd(B,C,Amp) & E divides A by H2,L0;
H8: E divides B & E divides C & E divides A by H7,L4;
H9: E divides C & E divides gcd(A,B,Amp) by H8,Def4;
H10: E divides D by H1,H9,Def4;
H11: D is_associated_to E by H6,H10,Def3;
H12: D is Element of Amp by H1,Def4;
H13: E is Element of Amp by H2,Def4;
H14: D = E by H11,H12,H13,AMP;
thus thesis by H1,H2,H14;
end;
```

◇

Definition referenced in part 25.

⟨proof of theorem T1 151b⟩ ≡

```
proof
let Amp be AmpleSet of I;
let A,B,C be Element of the carrier of I;
M: now per cases;
case A: C <> 0.I;
        consider D being Element of the carrier of I such that
        H1: D = gcd(A,B,Amp);
        K: now per cases;
        case A1: D <> 0.I;
          consider E being Element of the carrier of I such that
          H2: E = gcd(A*C,B*C,Amp);
          H3: D divides A & D divides B by H1,Def4;
          H4: C*D divides A*C & C*D divides B*C by H3,L3;
          H5: C*D divides gcd(A*C,B*C,Amp) by H4,Def4;
          H6: C*D divides E by H5,H2;
          consider F being Element of the carrier of I such that
          H7: E = (C*D)*F by H6,Def1;
          H8: E divides A*C &  E divides B*C by H2,L0;
          H10: (C*D)*F divides A*C & (C*D)*F divides B*C by H8,H7;
          H12: D*F divides A & D*F divides B
               proof
               consider G being Element of the carrier of I such that
               M1: ((C*D)*F)*G = A*C by H10,Def1;
```

```
            M2: (C*(D*F))*G = ((C*D)*F)*G  by VECTSP_1:def 16
                           .= C*A          by M1;
            M3: C*(D*F) divides C*A by M2,Def1;
            M4: D*F divides A by M3,L9,A;
            consider G being Element of the carrier of I such that
            M5: ((C*D)*F)*G = B*C by H10,Def1;
            M6: (C*(D*F))*G = ((C*D)*F)*G  by VECTSP_1:def 16
                           .= C*B          by M5;
            M7: C*(D*F) divides C*B by M6,Def1;
            M8: D*F divides B by M7,L9,A;
            thus thesis by M4,M8;
            end;
          H13: D*F divides gcd(A,B,Amp) by H12,Def4;
          H14: D*F divides D by H13,H1;
          H15: F divides 1.I
            proof
            M1: D = D*1.I by VECTSP_2:1;
            M2: D*F divides D*1.I by M1,H14;
            thus thesis by M2,L9,A1;
            end;
          H16: F is_unit by H15,Def2;
          H18: ex f being Element of the carrier of I
            st (f is_unit & (C*D)*f = E) by H7,H16;
          H19: C*D is_associated_to E by H18,L11;
          H20: E is_associated_to C*D by H19,L2;
          thus gcd(A*C,B*C,Amp) is_associated_to C*gcd(A,B,Amp)
            by H20,H1,H2;
        case A2: D = 0.I;
          N1: gcd(A,B,Amp) = 0.I by A2,H1;
          N2: A = 0.I & B = 0.I by N1,L12;
          N3: C*gcd(A,B,Amp) = 0.I by N1,VECTSP_2:26;
          N4:   gcd(A*C,B*C,Amp)
              = gcd(0.I*C,0.I*C,Amp)  by N2
             .= gcd(0.I,0.I*C,Amp)    by VECTSP_2:26
             .= gcd(0.I,0.I,Amp)      by VECTSP_2:26
             .= 0.I                   by GCD0
             .= C*gcd(A,B,Amp)        by N3;
          N5:   gcd(A*C,B*C,Amp)*1.I
              = gcd(A*C,B*C,Amp)     by VECTSP_2:1
             .= C*gcd(A,B,Amp)       by N4;
          N6: gcd(A*C,B*C,Amp) divides C*gcd(A,B,Amp) by N5,Def1;
          N7:   (C*gcd(A,B,Amp))*1.I
              = C*gcd(A,B,Amp)       by VECTSP_2:1
             .= gcd(A*C,B*C,Amp)     by N4;
          N8: C*gcd(A,B,Amp) divides gcd(A*C,B*C,Amp) by N7,Def1;
          thus gcd(A*C,B*C,Amp) is_associated_to C*gcd(A,B,Amp)
            by Def3,N6,N8;
        end;  :: cases K
        thus gcd(A*C,B*C,Amp) is_associated_to C*gcd(A,B,Amp) by K;
case B: C = 0.I;
        H1: A*C = 0.I by B,VECTSP_2:26;
        H2: B*C = 0.I by B,VECTSP_2:26;
        H3:   gcd(A*C,B*C,Amp)
            = gcd(0.I,0.I,Amp)  by H1,H2
```

```
            .= 0.I               by GCD0
            .= 0.I*gcd(A,B,Amp)  by VECTSP_2:26
            .= C*gcd(A,B,Amp)    by B;
         H4:   gcd(A*C,B*C,Amp)*1.I
             = gcd(A*C,(B*C),Amp)   by VECTSP_2:1
            .= C*gcd(A,B,Amp)       by H3;
         H5: gcd(A*C,B*C,Amp) divides C*gcd(A,B,Amp) by H4,Def1;
         H6:   (C*gcd(A,B,Amp))*1.I
             = C*gcd(A,B,Amp)       by VECTSP_2:1
            .= gcd(A*C,B*C,Amp)     by H3;
         H7: C*gcd(A,B,Amp) divides gcd(A*C,B*C,Amp) by H6,Def1;
         thus gcd(A*C,B*C,Amp) is_associated_to C*gcd(A,B,Amp)
              by H5,H7,Def3;
end; :: cases M
thus thesis by M;
end;
```

◇

Definition referenced in part 25.

⟨proof of theorem T2 153⟩ ≡

```
proof
let Amp be AmpleSet of I;
let A,B,C be Element of the carrier of I;
assume H1: gcd(A,B,Amp) = 1.I;
H2: gcd(A*C,B*C,Amp) is_associated_to C*gcd(A,B,Amp) by T1;
H3: C*gcd(A,B,Amp) = C*1.I  by H1
                  .= C      by VECTSP_2:1;
H4: gcd(A*C,B*C,Amp) is_associated_to C by H2,H3;
H5: C is_associated_to gcd(A*C,B*C,Amp) by H4,L2;
H6: gcd(A,C,Amp) is_associated_to gcd(A,gcd(A*C,B*C,Amp),Amp)
    by H5,L14;
H7a: gcd(A,gcd(A*C,B*C,Amp),Amp) =
     gcd(gcd(A,A*C,Amp),B*C,Amp) by T0;
H7: gcd(A,gcd(A*C,B*C,Amp),Amp) is_associated_to
    gcd(gcd(A,A*C,Amp),B*C,Amp) by H7a,L2;
H8: gcd(A,C,Amp) is_associated_to gcd(gcd(A,A*C,Amp),B*C,Amp)
    by H6,H7,L2;
H9: gcd(A,A*C,Amp) is_associated_to A
    proof
    M1: A = A*1.I & A is_associated_to A by L2,VECTSP_2:1;
    M2: A is_associated_to A*1.I by M1;
    M3: gcd(A,A*C,Amp) is_associated_to gcd(A*1.I,A*C,Amp) by M2,L14;
    M4: gcd(A*1.I,A*C,Amp) is_associated_to A*gcd(1.I,C,Amp) by T1;
    M5: A*gcd(1.I,C,Amp) = A*1.I  by GCD2
                         .= A      by VECTSP_2:1;
    M6: gcd(A*1.I,A*C,Amp) is_associated_to A by M5,M4;
    thus thesis by M6,M3,L2;
    end;
H10: gcd(gcd(A,A*C,Amp),B*C,Amp) is_associated_to gcd(A,B*C,Amp)
     by H9,L14;
H11: gcd(A,C,Amp) is_associated_to gcd(A,B*C,Amp) by H8,H10,L2;
H12: gcd(A,B*C,Amp) is_associated_to gcd(A,C,Amp) by H11,L2;
H13: gcd(A,B*C,Amp) is Element of Amp by Def4;
H14: gcd(A,C,Amp) is Element of Amp by Def4;
H15: gcd(A,B*C,Amp) = gcd(A,C,Amp) by H12,H13,H14,AMP;
```

```
    thus thesis by H15;
    end;
```
◇
Definition referenced in part 25.

⟨proof of theorem T4 154⟩ ≡
```
    proof
    let Amp be AmpleSet of I;
    let A,B,C be Element of the carrier of I;
    consider D being Element of the carrier of I such that
    H1: D = gcd(A,C,Amp);
    H2: D divides A & D divides C by H1,Def4;
    H2b: D is Element of Amp by H1,Def4;
    consider E being Element of the carrier of I such that
    H3: D*E = A by H2,Def1;
    consider F being Element of the carrier of I such that
    H4: D*F = C by H2,Def1;
    H5: D divides A+B*C
        proof
        M1: D*(E+F*B) = D*E+D*(F*B)  by VECTSP_2:1
                     .= D*E+(D*F)*B  by VECTSP_1:def 16
                     .= A+B*C        by H3,H4;
        thus thesis by M1,Def1;
        end;
    H6: for z being Element of the carrier of I
        st (z divides A+B*C & z divides C)
        holds z divides D
        proof
        let Z be Element of the carrier of I;
        assume M1: Z divides A+B*C & Z divides C;
        M1a: Z divides C by M1;
        consider X being Element of the carrier of I such that
        M2: Z*X = C by M1,Def1;
        consider Y being Element of the carrier of I such that
        M3: Z*Y = A+B*C by M1,Def1;
        M4:   Z*(Y+(-(B*X)))
            = Z*Y+Z*(-(B*X))    by VECTSP_2:1
           .= Z*Y+(-(Z*(X*B)))  by VECTSP_2:28
           .= Z*Y+(-((Z*X)*B))  by VECTSP_1:def 16
           .= (A+B*C)+(-(C*B))  by M2,M3
           .= A+(B*C+(-(C*B)))  by VECTSP_2:1
           .= A+0.I             by VECTSP_2:1
           .= A                 by VECTSP_2:1;
        M5: Z divides A by M4,Def1;
        M6: Z divides D by M1a,M5,H1,Def4;
        thus thesis by M6;
        end;
    H7: D = gcd(A+B*C,C,Amp) by H2,H2b,H5,H6,Def4;
    thus thesis by H1,H7;
    end;
```
◇
Definition referenced in part 25.

A.4 Lemmata about Fractions

We start with the file QF.VOC which intorduces new vocabulary items for fractions and their constructors.

"qf.voc" 156a ≡

```
MFraction
MFractions
R~
Ris_normalized_wrt
Onum
Odenom
Ofract
```

◇

The rest of this section contains some remaining proofs about fractions over integral domains, as well as the definition of fraction multiplication with the corresponding theorem concerning the multiplicative unity of fractions.

⟨proof of fraction's constructor equation 156b⟩ ≡

```
proof
let I be domRing;
let u be Fraction of I;
consider a,b being Element of the carrier of I such that
H1: u = [a,b] & b <> 0.I by Def52;
H2: fract(a,b) = [a,b] by Def54,H1
                .= u     by H1;
H3: a = u`1    by H1,MCART_1:def 1
     .= num(u) by Def55;
H4: b = u`2      by H1,MCART_1:def 2
     .= denom(u) by Def53;
thus thesis by H2,H3,H4;
end;
```

◇

Definition referenced in part 79a.

⟨proof of denom 156c⟩ ≡

```
proof
let I be domRing;
let u be Fraction of I;
H0: u`2 <> 0.I by N;
thus thesis by H0,Def53;
end;
```

◇

Definition referenced in part 79b.

⟨proof of F1 157a⟩ ≡

```
proof
let I be domRing;
let u be Fraction of I ;
let a,b be Element of the carrier of I;
assume H0: b <> 0.I;
H1: fract(a,b) = u implies (a = num(u) & b = denom(u))
    proof

    assume M1: fract(a,b) = u;
```

```
        M2: u =[a,b] by M1,Def54,H0;
        M3: num(u) = u‘1  by Def55
                  .= a    by M2,MCART_1:def 1;
        M4: denom(u) = u‘2  by Def53
                    .= b    by M2,MCART_1:def 2;
        thus thesis by M3,M4;
        end;
    H2: (a = num(u) & b = denom(u)) implies fract(a,b) = u
        proof
        assume M1: a = num(u) & b = denom(u);
        consider a’,b’ being Element of the carrier of I such that
        M6: u = [a’,b’] & b’ <> 0.I by Def52;
        M3: a’ = u‘1  by M6,MCART_1:def 1
              .= a    by M1,Def55;
        M4: b’ = u‘2  by M6,MCART_1:def 2
              .= b    by M1,Def53;
        M5: u = [a,b] by M6,M3,M4;
        thus thesis by H0,Def54,M5;
        end;
    thus thesis by H1,H2;
    end;
```

◇

Definition referenced in part 80a.

⟨proof of F2 157b⟩ ≡

```
    proof
    let I be domRing;
    let r,s be Fraction of I;
    let r1,r2,s1,s2 be Element of the carrier of I;
    assume H0: r1 = num(r) & r2 = denom(r) & s1 = num(s) & s2 = denom(s);
    H3: r+s = [r‘1*s‘2+s‘1*r‘2, r‘2*s‘2]  by Def70
           .= [r‘1*s2+s‘1*r‘2, r‘2*s‘2]   by H0,Def53
           .= [r‘1*s2+s‘1*r‘2, r‘2*s2]    by H0,Def53
           .= [r‘1*s2+s‘1*r‘2, r2*s2]     by H0,Def53
           .= [r‘1*s2+s‘1*r2, r2*s2]      by H0,Def53
           .= [r1*s2+s‘1*r2, r2*s2]       by H0,Def55
           .= [r1*s2+s1*r2, r2*s2]        by H0,Def55;
    H4: num(r+s) = (r+s)‘1 & denom(r+s) = (r+s)‘2 by Def53,Def55;
    thus thesis by H3,H4,MCART_1:def 1,MCART_1:def 2;
    end;
```

◇

Definition referenced in part 81b.

⟨proof of fraction addition 158a⟩ ≡

```
    proof
    let I be domRing;
    let u,v be Fraction of I;
    H0: u+v = [u‘1*v‘2+v‘1*u‘2,u‘2*v‘2] by Def70
           .= [num(u)*v‘2+v‘1*u‘2,u‘2*v‘2] by Def55
           .= [num(u)*v‘2+num(v)*u‘2,u‘2*v‘2] by Def55
           .= [num(u)*denom(v)+num(v)*u‘2,u‘2*v‘2] by Def53
           .= [num(u)*denom(v)+num(v)*u‘2,u‘2*denom(v)] by Def53
           .= [num(u)*denom(v)+num(v)*denom(u),u‘2*denom(v)] by Def53
           .= [num(u)*denom(v)+num(v)*denom(u),denom(u)*denom(v)] by Def53;
    H1: denom(u) <> 0.I & denom(v) <> 0.I by TT;
```

```
    H2: denom(u)*denom(v) <> 0.I by H1,VECTSP_2:15;
    thus thesis by H0,H2,Def54;
    end;
◇
```
Definition referenced in part 81a.

What follows is the definition of fraction multiplication. Note that it is almost the same as the corresponding definition for fraction addition.

"BrHenAdd.miz" 158b ≡
```
    definition
    let I be domRing;
    let u,v be Fraction of I;
    func u*v -> Fraction of I means :Def80:
     it = [u‘1*v‘1,u‘2*v‘2];
    existence
     proof
     H1: u‘2 <> 0.I & v‘2 <> 0.I by N;
     H2: u‘2*v‘2 <> 0.I by H1,VECTSP_2:15;
     consider a being Element of the carrier of I such that
     H6: a = u‘1*v‘1;
     consider b being Element of the carrier of I such that
     H7: b = u‘2*v‘2;
     consider u being Element of [:the carrier of I,the carrier of I:]
     such that H3: u = [a,b];
     H5: ex a,b being Element of the carrier of I st
         u = [a,b] & b <> 0.I by H3,H2,H7;
     H4: u is Fraction of I by H5,Def52;
     thus thesis by H3,H4,H6,H7;
     end;
    uniqueness;
    end;

    theorem
    for I being domRing
    for u,v being Fraction of I holds
    u*v = [num(u)*num(v),denom(u)* denom(v)]
    proof
    let I be domRing;
    let u,v be Fraction of I;
    H0: u*v = [u‘1*v‘1,u‘2*v‘2] by Def80
           .= [num(u)*v‘1,u‘2*v‘2] by Def55
           .= [num(u)*num(v),u‘2*v‘2] by Def55
           .= [num(u)*num(v),u‘2*denom(v)] by Def53
           .= [num(u)*num(v),denom(u)*denom(v)] by Def53;
    thus thesis by H0;
    end;
◇
```
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We conclude with stating and proving theorems about the additive and multiplicative unity of fractions. The proofs are easy done by substituting the definition of `0.Q` resp. `1.Q` in the definition of fraction addition resp. fraction multiplication.

"BrHenAdd.miz" 159 ≡

```
    theorem
    for I being domRing
    for Q being Fractions of I
    for u being Fraction of I holds
    u + 0.Q = u & 0.Q + u = u
    proof
    let I be domRing;
    let Q be Fractions of I;
    let u be Fraction of I;
    H0: 0.I <> 1.I by VECTSP_1:31;
    H1: (0.Q)`1 = (fract(0.I,1.I))`1  by Def74
              .= [0.I,1.I]`1          by H0,Def54
              .= 0.I                  by MCART_1:def 1;
    H2: (0.Q)`2 = (fract(0.I,1.I))`2  by Def74
              .= [0.I,1.I]`2          by H0,Def54
              .= 1.I                  by MCART_1:def 2;
    consider a,b being Element of the carrier of I such that
    H3: u = [a,b] & b <> 0.I by Def52;
    H4: a = u`1 & b = u`2 by H3,MCART_1:def 1,MCART_1:def 2;
    H5: u+0.Q = [u`1*(0.Q)`2+(0.Q)`1*u`2,u`2*(0.Q)`2] by Def70
            .= [u`1*(0.Q)`2+0.I*u`2,u`2*(0.Q)`2]     by H1
            .= [u`1*1.I+0.I*u`2,u`2*1.I]             by H2
            .= [u`1*1.I+0.I,u`2*1.I]                 by VECTSP_2:26
            .= [u`1*1.I,u`2*1.I]                     by VECTSP_2:1
            .= [u`1,u`2*1.I]                         by VECTSP_2:1
            .= [u`1,u`2]                             by VECTSP_2:1;
    H6: 0.Q+u = [(0.Q)`1*u`2+u`1*(0.Q)`2,(0.Q)`2*u`2] by Def70
            .= [0.I*u`2+u`1*(0.Q)`2,(0.Q)`2*u`2]     by H1
            .= [0.I*u`2+u`1*1.I,1.I*u`2]             by H2
            .= [0.I+u`1*1.I,1.I*u`2]                 by VECTSP_2:26
            .= [u`1*1.I,1.I*u`2]                     by VECTSP_2:1
            .= [u`1,u`2*1.I]                         by VECTSP_2:1
            .= [u`1,u`2]                             by VECTSP_2:1;
    thus thesis by H3,H4,H5,H6;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

"BrHenAdd.miz" 160 ≡

```
    theorem
    for I being domRing
    for Q being Fractions of I
    for u being Fraction of I holds
    u * 1.Q = u & 1.Q * u = u
    proof
    let I be domRing;
    let Q be Fractions of I;
    let u be Fraction of I;
    H0: 0.I <> 1.I by VECTSP_1:31;
    H1: (1.Q)`1 = (fract(1.I,1.I))`1  by Def75
              .= [1.I,1.I]`1          by H0,Def54
              .= 1.I                  by MCART_1:def 1;
    H2: (1.Q)`2 = (fract(1.I,1.I))`2  by Def75
```

```
            .= [1.I,1.I]`2          by H0,Def54
            .= 1.I                  by MCART_1:def 2;
    consider a,b being Element of the carrier of I such that
    H3: u = [a,b] & b <> 0.I by Def52;
    H4: a = u`1 & b = u`2 by H3,MCART_1:def 1,MCART_1:def 2;
    H5: u*1.Q = [u`1*(1.Q)`1,u`2*(1.Q)`2]  by Def80
             .= [u`1*1.I,u`2*(1.Q)`2]      by H1
             .= [u`1*1.I,u`2*1.I]          by H2
             .= [u`1,u`2*1.I]              by VECTSP_2:1
             .= [u`1,u`2]                  by VECTSP_2:1;
    H6: 1.Q*u = [(1.Q)`1*u`1,(1.Q)`2*u`2]  by Def80
             .= [1.I*u`1,(1.Q)`2*u`2]      by H1
             .= [1.I*u`1,1.I*u`2]          by H2
             .= [u`1,u`2*1.I]              by VECTSP_2:1
             .= [u`1,u`2]                  by VECTSP_2:1;
    thus thesis by H3,H4,H5,H6;
    end;
```

◇

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

A.5 Remaining Verification Conditions

Here we list the verification conditions for generic Brown/Henrici addition we left out in section 4.1. Note again that these theorems are automatically constructed by our verification condition generator.

We start with the theorems directly connected to the output `t` of the generic Brown/ Henrici addition algorithm.

"BrHenAdd-theorems.txt" 161 ≡

```
(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
              (= r 0) (= t s))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
              (not (= r 0)) (= s 0) (= t r))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
              (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
              (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
              (not (and (= r2 1) (= s2 1))) (= r2 1)
              (= t (fract (+ (* r1 s2) s1) s2)))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
              (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
              (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
              (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (= s2 1)
              (= t (fract (+ (* s1 r2) r1) r2)))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))
```

```
(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (not (= d 1))
           (= r2' (/ r2 d)) (= s2' (/ s2 d)) (= t1 0) (= t 0)
           (= t1 (+ (* r1 s2') (* s1 r2'))) (= t2 (* r2 s2')))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (not (= d 1)) (= r2' (/ r2 d))
           (= s2' (/ s2 d)) (= t1 (+ (* r1 s2') (* s1 r2')))
           (= t2 (* r2 s2')) (not (= t1 0)) (\in e Amp) (= e (gcd t1 d))
           (= t1' (/ t1 e)) (= t2' (/ t2 e)) (= t (fract t1' t2')))
         (and (~ t (+ r s)) (is-normalized-wrt t Amp)))
```

◇

File defined by parts 59ab, 161, 162.

The following theorems are due to the procedure calls of the Brown/Henrici algorithm, namely due to the `fract` and the `/` function. They state that these calls are correct with respect to the input specification of the corresponding subalgorithms.

"BrHenAdd-theorems.txt" 162 ≡

```
(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (= r2 1) (= s2 1))
         (not (= 1 0)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (= d 1))
         (not (= (* r2 s2) 0)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (not (= d 1)))
         (and (not (= d 0)) (d divides r2)))

(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (not (= d 1)) (= r2' (/ r2 d)))
         (and (not (= d 0)) (d divides s2)))
```

```
(implies (and (is-normalized-wrt r Amp) (is-normalized-wrt s Amp)
           (not (= r 0)) (not (= s 0)) (= r1 (num r)) (= r2 (denom r))
           (not (= r2 0)) (= s1 (num s)) (= s2 (denom s)) (not (= s2 0))
           (not (and (= r2 1) (= s2 1))) (not (= r2 1)) (not (= s2 1))
           (\in d Amp) (= d (gcd r2 s2)) (not (= d 1)) (= r2' (/ r2 d))
           (= s2' (/ s2 d)) (= t1 (+ (* r1 s2') (* s1 r2')))
           (= t2 (* r2 s2')) (not (= t1 0)) (\in e Amp) (= e (gcd t1 d)))
         (and (not (= e 0)) (e divides t1)))
```

◇

File defined by parts 59ab, 161, 162.

For completion we also present the prototypes of the occurring subalgorithms, which are necessary for the verification condition generator to construct the above given theorems for generic Brown/Henrici addition.

"prototypes.txt" 163 ≡

```
(prototype
 (+ r s out t)
 (input (\in r I) (\in s I))
 (output (\in t I)
         (with (= t (+ r s)))))

(prototype
 (* r s out t)
 (input (\in r I) (\in s I))
 (output (\in t I)
         (with (= t (* r s)))))

(prototype
 (num r out r1)
 (input (\in r Q))
 (output (\in r1 I)
         (with (= r1 (num r))) ))

(prototype
 (denom r out r2)
 (input (\in r Q))
 (output (\in r2 I)
         (with (= r2 (denom r)) (not(= r2 0)) )) )

(prototype
 (fract r s out t)
 (input (\in r I) (\in s I)
        (with (not(= s 0)) ))
 (output (\in t Q) (with (= t (fract r s))) ))

(prototype
 (/ r s out t)
 (input (\in r I) (\in s I)
        (with (not(= s 0)) (s divides r) ))
 (output (\in t I) (with (= t (/ r s))) ))
```

◇

File defined by parts 42a, 163.

A.6 Proofs of the Remaining Verification Conditions

For completion we start with the necessary environment making the file `BrHenAdd.miz` to a correct MIZAR article accepted by the MIZAR proof checker.

⟨BrHenAdd environment 164⟩ ≡

```
environ

vocabulary
COORD,VECTSP_1,VECTSP_2,LINALG_1,REAL_1,GCD,QF;
notation
STRUCT_0,RLVECT_1,MCART_1,VECTSP_1,DOMAIN_1,ZFMISC_1,
VECTSP_2,GCD;
constructors
GCD,DOMAIN_1,VECTSP_1,ALGSTR_2;
theorems
TARSKI,MCART_1,VECTSP_1,VECTSP_2,GCD;
definitions
STRUCT_0;
clusters
STRUCT_0,ZFMISC_1;

begin

reserve X,Y,Z for set;
reserve I for domRing;
reserve a,b,c,d for Element of the carrier of I;
⟨lemmata for Brown/Henrici 131⟩
```

◇

Definition referenced in part 75a.

Finally, here we present the MIZAR proofs of the verification conditions we did not prove in section 4.5. Note that these proofs follow the same scheme as the ones presented in the text. We start with theorems concerning the output `t` of the algorithm. The main job is to show that the theorem of Brown and Henrici — if necessary at all — is applicable, thus proving that `t` again is normalized. Showing `t ~ r + s` is straightforward.

"BrHenAdd.miz" 165 ≡

```
theorem
(Amp is_multiplicative &
 r is_normalized_wrt Amp & s is_normalized_wrt Amp &
 r = 0.Q & t = s)
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
M: now assume
H0: s is_normalized_wrt Amp & r = 0.Q & t = s;
H1: 1.G <> 0.G by VECTSP_1:def 21;
H2: r = fract(0.G,1.G) by H0,Def74;
H4: 0.G = num(r) & 1.G = denom(r) by H2,H1,F1;
H5: r‘1 = 0.G by H4,Def55;
H6: r‘2 = 1.G by H4,Def53;
H3: r+s = [r‘1*s‘2+s‘1*r‘2, r‘2*s‘2]  by Def70
```

```
       .= [0.G*s`2+s`1*r`2, r`2*s`2]  by H5
       .= [0.G*s`2+s`1*1.G, 1.G*s`2]  by H6
       .= [0.G+s`1*1.G, 1.G*s`2]      by VECTSP_2:26
       .= [s`1*1.G, 1.G*s`2]          by VECTSP_2:1
       .= [s`1, 1.G*s`2]              by VECTSP_2:1
       .= [s`1, s`2]                  by VECTSP_2:1;
H4: num(r+s) = (r+s)`1  by Def55
            .= s`1      by H3,MCART_1:def 1;
H5: denom(r+s) = (r+s)`2  by Def53
              .= s`2      by H3,MCART_1:def 2;
H6: num(t)*denom(r+s) = s`1*denom(r+s)     by H0,Def55
                     .= s`1*s`2            by H5
                     .= num(r+s)* s`2      by H4
                     .= num(r+s)*denom(t)  by H0,Def53;
H7: t ~ (r+s) by H6,Def76;
thus thesis by H7;
end;  :: M
thus thesis by M;
end;

theorem
(Amp is_multiplicative &
 r is_normalized_wrt Amp & s is_normalized_wrt Amp &
 not(r = 0.Q) & s = 0.Q & t = r)
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
M: now assume
H0: r is_normalized_wrt Amp & s = 0.Q & t = r;
H1: 1.G <> 0.G by VECTSP_1:def 21;
H2: s = fract(0.G,1.G) by H0,Def74;
H4: 0.G = num(s) & 1.G = denom(s) by H2,H1,F1;
H5: s`1 = 0.G by H4,Def55;
H6: s`2 = 1.G by H4,Def53;
H3: r+s = [r`1*s`2+s`1*r`2, r`2*s`2]  by Def70
       .= [r`1*s`2+0.G*r`2, r`2*s`2]  by H5
       .= [r`1*1.G+0.G*r`2, r`2*1.G]  by H6
       .= [r`1*1.G+0.G, r`2*1.G]      by VECTSP_2:26
       .= [r`1*1.G, r`2*1.G]          by VECTSP_2:1
       .= [r`1, r`2*1.G]              by VECTSP_2:1
       .= [r`1, r`2]                  by VECTSP_2:1;
H4: num(r+s) = (r+s)`1  by Def55
            .= r`1      by H3,MCART_1:def 1;
H5: denom(r+s) = (r+s)`2  by Def53
              .= r`2      by H3,MCART_1:def 2;
H6: num(t)*denom(r+s) = r`1*denom(r+s)     by H0,Def55
                     .= r`1*r`2            by H5
                     .= num(r+s)* r`2      by H4
                     .= num(r+s)*denom(t)  by H0,Def53;
H7: t ~ (r+s) by H6,Def76;
thus thesis by H7;
end;  :: M
thus thesis by M;
end;
```

```
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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 = 1.G &
 t = fract(r1*s2+s1,s2))
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
M: now assume
H0: s is_normalized_wrt Amp & r1 = num(r) & r2 = denom(r) &
    s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
    r2 = 1.G & t = fract(r1*s2+s1,s2);
H1b: s2 ∈ Amp by H0,Def73;
H1: denom(t) = s2 by H0,F1;
H2: num(t) = r1*s2+s1 by H0,F1;
H3: denom(t) ∈ Amp by H1b,H1;
H4:   gcd(r1*s2+s1,s2,Amp)
    = gcd(s1,s2,Amp)        by GCD:39
   .= 1.G                   by H0,Def73;
H5: t is_normalized_wrt Amp by H4,H3,H2,H1,Def73;

H7: num(r+s) = r1*s2+s1*r2   by H0,F2
            .= r1*s2+s1*1.G  by H0
            .= r1*s2+s1      by VECTSP_2:1;
H8: denom(r+s) = r2*s2    by H0,F2
               .= 1.G*s2   by H0
               .= s2       by VECTSP_2:1;
H9: num(t)*denom(r+s) = (r1*s2+s1)*denom(r+s)  by H2
                      .= (r1*s2+s1)*s2         by H8
                      .= num(r+s)*s2           by H7
                      .= num(r+s)*denom(t)     by H1;
H10: t ~ (r+s) by H9,Def76;

thus thesis by H10,H5;
end;  :: M
thus thesis by M;
end;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 = 1.G &
 t = fract(s1*r2+r1,r2))
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
M: now assume
H0: r is_normalized_wrt Amp & r1 = num(r) & r2 = denom(r) &
    r2 <> 0.G & s1 = num(s) & s2 = denom(s) &
```

```
    s2 = 1.G & t = fract(s1*r2+r1, r2);
H1b: r2 ∈ Amp by H0,Def73;
H1: denom(t) = r2 by H0,F1;
H2: num(t) = s1*r2+r1 by H0,F1;
H3: denom(t) ∈ Amp by H1b,H1;
H4:   gcd(s1*r2+r1,r2,Amp)
    = gcd(r1,r2,Amp)        by GCD:39
   .= 1.G                   by H0,Def73;
H5: t is_normalized_wrt Amp by H4,H3,H2,H1,Def73;

H7: num(r+s) = r1*s2+s1*r2   by H0,F2
            .= r1*1.G+s1*r2  by H0
            .= r1+s1*r2      by VECTSP_2:1;
H8: denom(r+s) = r2*s2    by H0,F2
              .= r2*1.G   by H0
              .= r2       by VECTSP_2:1;
H9: num(t)*denom(r+s) = (s1*r2+r1)*denom(r+s)  by H2
                     .= (s1*r2+r1)*r2          by H8
                     .= num(r+s)*r2            by H7
                     .= num(r+s)*denom(t)      by H1;
H10: t ~ (r+s) by H9,Def76;

thus thesis by H10,H5;
end;  :: M
thus thesis by M;
end;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
 r2' = r2/d & s2' = s2/d &
 t1 = r1*s2'+s1*r2' & t2 = r2*s2' & t1 = 0.G & t = 0.Q)
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
M: now assume
H0: 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 &
    r2' = r2/d & s2' = s2/d &
    t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
    t1 = 0.G & t = 0.Q;
H2a: 1.G <> 0.G by VECTSP_1:def 21;
H2: t = fract(0.G,1.G) by H0,Def74;
H4: 0.G = num(t) & 1.G = denom(t) by H2,H2a,F1;
H3: denom(t) ∈ Amp by H4,GCD:21;
H1: gcd(num(t),denom(t),Amp) = 1.G by H4,GCD:32;
H7: t is_normalized_wrt Amp by H1,H3,Def73;

H9: gcd(r2,s2,Amp) <> 0.G by H0,GCD:33;
```

```
H10a: gcd(r2,s2,Amp) divides r2 by GCD:27;
H10: gcd(r2,s2,Amp) divides s1*r2 by H10a,GCD:7;
H11a: gcd(r2,s2,Amp) divides s2 by GCD:27;
H11: gcd(r2,s2,Amp) divides r1*s2 by GCD:7,H11a;
H13: gcd(r2,s2,Amp) divides r1*s2+s1*r2 by H10,H11,L1;
H8: r1*s2+s1*r2 = 0.G
    proof
    M3:   0.G
        = r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)) by H0
       .= (r1*s2)/gcd(r2,s2,Amp)+s1*(r2/gcd(r2,s2,Amp)) by H9,H11a,L3
       .= (r1*s2)/gcd(r2,s2,Amp)+(s1*r2)/gcd(r2,s2,Amp) by H9,H10a,L3
       .= (r1*s2+s1*r2)/gcd(r2,s2,Amp) by H9,H11,H10,L2;
    thus thesis by H13,H9,M3,GCD:8;
    end;
H14: num(r+s) = 0.G by H8,H0,F2;
H16: num(t)*denom(r+s) = 0.G*denom(r+s)     by H4
                      .= 0.G                by VECTSP_2:26
                      .= 0.G*1.G            by VECTSP_2:26
                      .= num(r+s)*1.G       by H14
                      .= num(r+s)*denom(t)  by H4;
H17: t ~ (r+s) by H16,Def76;

thus thesis by H17,H7;
end;  :: M
thus thesis by M;
end;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
 r2' = r2/d & s2' = s2/d &
 t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
 t1 <> 0.G & e ∈ Amp & e = gcd(t1,d,Amp) &
 t1' = t1/e & t2' = t2/e & t = fract(t1',t2'))
implies (t ~ r+s & t is_normalized_wrt Amp)
proof
assume H0: 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 &
           s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
           not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
           d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
           r2' = r2/d & s2' = s2/d &
           t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
           t1 <> 0.G & e ∈ Amp & e = gcd(t1,d,Amp) &
           t1' = t1/e & t2' = t2/e & t = fract(t1',t2');
H1: t2' <> 0.G by H0,BH14;
H2: gcd(r1,r2,Amp) = 1.G & gcd(s1,s2,Amp) = 1.G by H0,Def73;
```

```
H4: t1' = num(t) & t2' = denom(t) by H0,H1,F1;
H5: num(t) = t1/gcd(t1,d,Amp)                              by H0,H4
          .= t1/gcd(r1*s2'+s1*r2',gcd(r2,s2,Amp),Amp)  by H0
          .= t1/gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
                    gcd(r2,s2,Amp),Amp) by H0
          .= (r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp))) /
             gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
                 gcd(r2,s2,Amp),Amp) by H0;
H6: denom(t) = t2/gcd(t1,d,Amp)                              by H0,H4
            .= t2/gcd(r1*s2'+s1*r2',gcd(r2,s2,Amp),Amp)  by H0
            .= t2/gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
                      gcd(r2,s2,Amp),Amp)   by H0
            .= (r2*(s2/gcd(r2,s2,Amp))) /
               gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
                      gcd(r2,s2,Amp),Amp)   by H0;
H7: gcd(r2,s2,Amp) <> 0.G by H0,GCD:33;
H8: gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
        gcd(r2,s2,Amp),Amp) <> 0.G  by H7,GCD:33;
H9:   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) by H0,H2,GCD:40;
H10:  gcd(num(t),denom(t),Amp) = 1.G by H5,H6,H8,H9,GCD:38;

H11: r2 ∈ Amp & s2 ∈ Amp by H0,Def73;
reconsider r2,s2 as Element of Amp by H11;
H12: gcd(r2,s2,Amp) ∈ Amp by GCD:def 12;
H13: gcd(r2,s2,Amp) divides s2 by GCD:def 12;
H18: gcd(r2,s2,Amp) <> 0.G by H0,GCD:33;
reconsider z1 = gcd(r2,s2,Amp) as Element of Amp by H12;
H14: s2/z1 ∈ Amp by H0,H13,H18,GCD:24;
reconsider z2 = s2/gcd(r2,s2,Amp) as Element of Amp by H14;
H15: r2*z2 ∈ Amp by H0,GCD:def 9;
reconsider z3 = r2*(s2/gcd(r2,s2,Amp)) as Element of Amp by H15;
reconsider z4 = gcd(r1*(s2/gcd(r2,s2,Amp))+s1*(r2/gcd(r2,s2,Amp)),
                    gcd(r2,s2,Amp),Amp) as Element of Amp by GCD:def 12;
H16: z4 <> 0.G by H18,GCD:33;
H23: gcd(r2,s2,Amp) divides r2 by GCD:def 12;
H17: z4 divides z3
     proof
     M1: z4 divides gcd(r2,s2,Amp) by GCD:def 12;
     M3: z4 divides r2 by M1,H23,GCD:2;
     thus thesis by M3,GCD:7;
     end;
H19: z3/z4 ∈ Amp by H0,H16,H17,GCD:24;
H20: denom(t) ∈ Amp by H19,H6;
H21: t is_normalized_wrt Amp by H20,H10,Def73;

H24: gcd(r2,s2,Amp) divides r1*s2 by H13,GCD:7;
H27: gcd(r2,s2,Amp) divides s1*r2 by H23,GCD:7;
H28: gcd(r2,s2,Amp) divides ((r1*s2)*r2) by H24,GCD:7;
H29: gcd(r2,s2,Amp) divides ((s1*r2)*r2) by H27,GCD:7;
H32: ((r1*(s2/gcd(r2,s2,Amp))) +
      (s1*(r2/gcd(r2,s2,Amp))))*(r2*s2)
```

```
    = ((r1*(s2/gcd(r2,s2,Amp)))*(r2*s2)) +
      ((s1*(r2/gcd(r2,s2,Amp)))*(r2*s2))  by VECTSP_2:1
   .= (((r1*s2)/gcd(r2,s2,Amp))*(r2*s2)) +
      ((s1*(r2/gcd(r2,s2,Amp)))*(r2*s2))  by H18,H13,L3
   .= (((r1*s2)/gcd(r2,s2,Amp))*(r2*s2)) +
      (((s1*r2)/gcd(r2,s2,Amp))*(r2*s2))  by H18,H23,L3
   .= ((((r1*s2)/gcd(r2,s2,Amp))*r2)*s2) +
      (((s1*r2)/gcd(r2,s2,Amp))*(r2*s2))  by VECTSP_1:def 16
   .= ((((r1*s2)/gcd(r2,s2,Amp))*r2)*s2) +
      ((((s1*r2)/gcd(r2,s2,Amp))*r2)*s2)  by VECTSP_1:def 16
   .= ((((r1*s2)*r2)/gcd(r2,s2,Amp))*s2) +
      ((((s1*r2)/gcd(r2,s2,Amp))*r2)*s2)  by H18,H24,L3
   .= ((((r1*s2)*r2)/gcd(r2,s2,Amp))*s2) +
      ((((s1*r2)*r2)/gcd(r2,s2,Amp))*s2)  by H18,H27,L3
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((((s1*r2)*r2)/gcd(r2,s2,Amp))*s2)  by H18,H28,L3
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((((s1*r2)*r2)*s2)/gcd(r2,s2,Amp))  by H18,H29,L3;
H33a: gcd(r2,s2,Amp) divides (r2*s2) by H23,GCD:7;
H33: gcd(r2,s2,Amp) divides ((r2*s2)*r1) by H33a,GCD:7;
H34: gcd(r2,s2,Amp) divides ((r2*s2)*s1) by H33a,GCD:7;
H37:   (r2*(s2/gcd(r2,s2,Amp))) *((r1*s2)+(s1*r2))
    = ((r2*(s2/gcd(r2,s2,Amp)))*(r1*s2)) +
      ((r2*(s2/gcd(r2,s2,Amp)))*(s1*r2))  by VECTSP_2:1
   .= (((r2*(s2/gcd(r2,s2,Amp)))*r1)*s2) +
      ((r2*(s2/gcd(r2,s2,Amp)))*(s1*r2))  by VECTSP_1:def 16
   .= (((r2*(s2/gcd(r2,s2,Amp)))*r1)*s2) +
      ((r2*(s2/gcd(r2,s2,Amp)))*s1)*r2)  by VECTSP_1:def 16
   .= ((((r2*s2)/gcd(r2,s2,Amp))*r1)*s2) +
      ((r2*(s2/gcd(r2,s2,Amp)))*s1)*r2)  by H18,H13,L3
   .= ((((r2*s2)/gcd(r2,s2,Amp))*r1)*s2) +
      ((((r2*s2)/gcd(r2,s2,Amp))*s1)*r2)  by H18,H13,L3
   .= ((((r2*s2)*r1)/gcd(r2,s2,Amp))*s2) +
      ((((r2*s2)/gcd(r2,s2,Amp))*s1)*r2)  by H18,H33a,L3
   .= ((((r2*s2)*r1)/gcd(r2,s2,Amp))*s2) +
      ((((r2*s2)*s1)/gcd(r2,s2,Amp))*r2)  by H18,H33a,L3
   .= ((((r2*s2)*r1)*s2)/gcd(r2,s2,Amp)) +
      ((((r2*s2)*s1)/gcd(r2,s2,Amp))*r2)  by H18,H33,L3
   .= ((((r2*s2)*r1)*s2)/gcd(r2,s2,Amp)) +
      ((((r2*s2)*s1)*r2)/gcd(r2,s2,Amp))  by H18,H34,L3
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((((r2*s2)*s1)*r2)/gcd(r2,s2,Amp))  by VECTSP_1:def 16
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((s1*((r2*s2)*r2))/gcd(r2,s2,Amp))  by VECTSP_1:def 16
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((s1*((r2*r2)*s2))/gcd(r2,s2,Amp))  by VECTSP_1:def 16
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      (((s1*(r2*r2))*s2)/gcd(r2,s2,Amp))  by VECTSP_1:def 16
   .= ((((r1*s2)*r2)*s2)/gcd(r2,s2,Amp)) +
      ((((s1*r2)*r2)*s2)/gcd(r2,s2,Amp))  by VECTSP_1:def 16;

H38:  ((r1*(s2/gcd(r2,s2,Amp))) +
        (s1*(r2/gcd(r2,s2,Amp))))*(r2*s2)
```

```
        = (r2*(s2/gcd(r2,s2,Amp)))*((r1*s2)+(s1*r2)) by H32,H37;
    H39: gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
             gcd(r2,s2,Amp),Amp)
         <> (0.G) by GCD:33,H18;
    H40: gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
             gcd(r2,s2,Amp),Amp) divides
         ((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))))
         by GCD:def 12;
    H42: gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
             gcd(r2,s2,Amp),Amp)
         divides gcd(r2,s2,Amp) by GCD:def 12;
    H43: gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
             gcd(r2,s2,Amp),Amp)
         divides r2 by H23,H42,GCD:2;
    H44: gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
             gcd(r2,s2,Amp),Amp)
         divides (r2*(s2/gcd(r2,s2,Amp))) by H43,GCD:7;
    H46:    (((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp)))) /
             gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
                 gcd(r2,s2,Amp),Amp))
            *(r2*s2)
         =  (((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp)))) *
              (r2*s2)) /
            gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
                gcd(r2,s2,Amp),Amp) by H39,H40,L3
        .=  ((r2*(s2/gcd(r2,s2,Amp)))*((r1*s2)+(s1*r2))) /
            gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
                gcd(r2,s2,Amp),Amp) by H38
        .= ((r2*(s2/gcd(r2,s2,Amp))) /
           gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
               gcd(r2,s2,Amp),Amp))
           *((r1*s2)+(s1*r2)) by H39,H44,L3;
    H49:   num(t)*denom(r+s)
         = num(t)*(r2*s2) by H0,F2
        .= (((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp)))) /
            gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
                gcd(r2,s2,Amp),Amp))
           *(r2*s2) by H5
        .= ((r2*(s2/gcd(r2,s2,Amp))) /
           gcd((r1*(s2/gcd(r2,s2,Amp)))+(s1*(r2/gcd(r2,s2,Amp))),
               gcd(r2,s2,Amp),Amp))
          *((r1*s2)+(s1*r2)) by H46
        .= denom(t)*((r1*s2)+(s1*r2)) by H6
        .= denom(t)*num(r+s) by H0,F2;
    H50: t ~ (r+s) by H49,Def76;

    thus thesis by H21,H50;
    end;
◇
```

File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

We conclude with the remaining theorems concerning correctness of the occurring procedure calls, namely of the subalgorithms / and `fract`. Note that all proofs are

done by simply referencing a suitable theorem.

"BrHenAdd.miz" 174 ≡

```
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 &
 s1 = num(s) & s2 = denom(s) & s2 = 0.G &
 r2  = 1.G & s2 = 1.G)
implies not(1.Q = 0.Q) by VECTSP_1:def 21;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d = 1.G)
implies r2*s2 <> 0.G by VECTSP_2:15;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G)
implies (d <> 0.G & d divides r2) by GCD:def 12,GCD:33;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G & r2' = r2/d)
implies (d <> 0.G & d divides s2) by GCD:def 12,GCD:33;

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 &
 s1 = num(s) & s2 = denom(s) & s2 <> 0.G &
 not(r2 = 1.G & s2 = 1.G) & r2 <> 1.G & s2 <> 1.G &
 d ∈ Amp & d = gcd(r2,s2,Amp) & d <> 1.G &
 r2' = r2/d & s2' = s2/d &
 t1 = r1*s2'+s1*r2' & t2 = r2*s2' &
 t1 <> 0.G & e ∈ Amp & e = gcd(t1,d,Amp))
implies (e <> 0.G & e divides t1) by GCD:def 12,GCD:33;
```

◇
File defined by parts 75ab, 76a, 77ab, 78, 79ab, 80ab, 81ab, 82ab, 83, 84ab, 85abc, 86ab, 87ab, 88abc, 89, 90abc, 91, 158b, 159, 160, 165, 174.

Appendix B

Additional SCHEME Code

This chapter contains further SCHEME code of the verification condition generator we did not include in the description in chapter three. Note that listing the code here enables the extraction of the generator's complete source code from this document using STWEB.

B.1 Some Further Main Functions

We start with some of the more important procedures the generator uses. Procedure `insert-pred-before` is a direct counterpart of `insert-pred-after` we presented in section 3.3. We use it to annotate sequences.

"annotations.scm" 177 ≡

```
(define (insert-pred-before symbol)
   (lambda (prog . theme)
      (if (equal? symbol 'all)
          (if (empty? prog)
              prog
              (if (equal? (car prog) 'begin)
                  (let ((rest
                         ((insert-pred-before 'all) (cddr prog))))
                     (append (list 'begin (cadr prog))
                             rest))
                   (let ((rest
                          ((insert-pred-before 'all) (cdr prog))))
                     (begin
                        (set! prednr (+ prednr 1))
                        (append (list prednr (car prog))
                                rest))) ))
          (do ((format (get formats (get-key prog))
                       (cdr format))
               (pr prog (cdr prog))
               (ergprog '() (append ergprog (list (car pr)) )))
              ((equal? (car format) symbol)
               (begin
                  (set! prednr (+ prednr 1))
                  (append ergprog (list prednr pr))))
```

```
                (if (empty? format)
                    (error 'insert-pred-before: symbol 'does 'not
                           'appear 'in 'format 'of prog)) )) ))
```

◇
File defined by parts 45a, 46b, 48b, 49a, 177.

Procedure `is-invariant?` is due to our rule concerning procedure calls: Together with procedure `is-free` it tests whether the given variables occur in the given formula.

"guesses.scm" 178 ≡

```
(define (is-invariant? formula vars)
   (cond ((empty? vars)  #t)
         ((is-free (car vars) formula) #f)
         (else
          (is-invariant? formula (cdr vars))) ))

(define (is-free obj formula)

   (define (is-free-h obj formula)
      (cond ((empty? formula)  #f)
            ((equal? formula obj) #t)
            ((and (not(list? formula))
                  (not(member formula logicals-list))
                  (not(equal? formula obj)))  #f)
            ((list? formula)
             (or (is-free-h obj (car formula))
                 (is-free-h obj (cdr formula))))
            (else #f) ))

   (let ((form (expand formula)))
      (if (equal? form '?) #t
          (is-free-h obj form))))
```

◇
File defined by parts 55ab, 56ac, 178.

Procedure `construct` is used in every stage of the generator. It gets an abstract scheme and another object — an algorithm, an annotated algorithm or a formula — as input. Out of the abstract scheme it builds the result by filling in certain parts of the scheme with the corresponding parts of the given object.
Because of its widespread use we describe this procedure in more detail: An abstract scheme consists of keywords. Some keywords — like `'proc`, `'inv` or the keywords contained in the vocabulary list `voc-list` — allow an immediate result. Keywords contained in `symbol-list` — `action` for instance — are replaced by using the format definition of the given object: Procedure `look-up` computes the object's statement corresponding to the given keyword. The third category of keywords consists of procedures contained in `construct-list` — for example `inputspec` or `outputparam`. These procedures are simply evaluated giving the desired result.
We use procedure `construct` for instance to compute current Hoare triples out of the abstract ones given by the activities.

"utilities.scm" 179 ≡
```
(define (construct obj scheme)
    (cond ((equal? scheme 'proc) obj)
          ((or (empty? scheme) (is-prednr? scheme)
               (member scheme key-list) (member scheme voc-list)
               (member scheme proc-list) (member scheme var-list)
               (member scheme logicals-list)) scheme)
          ((equal? scheme 'pre) (car obj))
          ((or (equal? scheme 'post)
               (equal? scheme 'intermed))
           (caddr obj))
          ((equal? scheme 'inv) (caddr (cadr obj)))
          ((equal? scheme 'first) (cadr obj))
          ((member scheme symbol-list) (lookup scheme obj))
          ((member scheme construct-list)
           (apply (eval (list scheme)) (list obj)))
          ((and (list? scheme)
                (member (car scheme) construct-list))
           (apply (eval scheme) (list obj)))
          ((list? scheme)
           (cons (construct obj (car scheme))
                 (construct obj (cdr scheme))))
          (else (error 'procedure 'construct:
                       scheme 'is 'unknown))))
```
◇
File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

Procedure `lookup` gets a symbol and a program as input. It takes the format definition of the given program and checks, whether the symbol appears within it. If the symbol is found, `lookup` returns the corresponding part of the program.

"utilities.scm" 180a ≡
```
(define (lookup symbol prog)

  (define (lookup-h symbol prog format)
     (do ((pr (if (is-annotated? prog) (cadr prog) prog)
              (if (and (not(empty? (cdr pr)))
                       (number? (cadr pr))
                       (not(empty? (cddr pr))))
                  (cddr pr) (cdr pr)))
          (form format (cdr form)))
         ((or (equal? (car form) symbol)
              (empty? (cdr form)))
          (if (equal? (car form) symbol)
              (car pr) #f)) ))

   (let ((format (get formats (get-key prog)) ))
      (if (equal? (car format) (get-key prog))
          (let ((result (lookup-h symbol prog format)))
             (if result result
                 (error 'procedure 'lookup: symbol
                        'does 'not 'appear 'in
                        (get formats (get-key prog)))))

          (do ((form format (cdr form)))
              ((fits? prog (car form))
                  (let ((result (lookup-h symbol prog (car form))))
                      (if result result
                          (error 'procedure 'lookup: symbol 'does
```

```
                                    'not 'appear 'in (car form)) )))
                  (if (empty? (cdr form))
                      (error 'procedure 'lookup: symbol
                             'does 'not 'appear 'in
                             (get formats (get-key prog))) )))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

Procedure `get-actual` looks for the format definition fitting to the given program. Note that for instance the *if*-statement has more than one possible format.

"utilities.scm" 180b ≡

```
(define (get-actual formats prog)
   (let ((format (get formats (get-key prog))))
     (if (list? (car format))
         (do ((forms format (cdr forms)))
             ((equal? (length (car forms)) (length prog))
              (car forms))
              (if (empty? forms)
                  (error 'procedure 'get-actual:
                         'no 'format 'for prog)))
         format)))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

To handle procedure calls we have to substitute formal parameters by actual parameters. In addition we need substitution for some rules of the Hoare calculus, for instance for the *assignment*-rule. We decided to separate stating that a substitution has to take place and actually executing a substitution. This allows the constructed abstract theorems to include terms like `(subst 0 (x y) (u v))` where `0` is an abstract predicate, hence to get a complete list of verification conditions independent of whether specific predicates can be constructed for the current algorithm.
Consequently, we get a procedure `subst` and a procedure `do-subst` — which does nothing more than executing substitutions in a given formula by applying procedure `subst`.

"utilities.scm" 181 ≡

```
(define (do-subst formula)
   (cond ((or (empty? formula)
              (not(list? formula))) formula)
         ((and (list? formula)
               (equal? (car formula) 'subst))
          (apply subst (do-subst (cdr formula))))
         (else  (cons (do-subst (car formula))
                      (do-subst (cdr formula)))) ))



(define (subst formula vars terms)

   (define (subst-h formula var term)
     (cond
       ((or (null? formula) (and (not(list? formula))
```

```
            (not(equal? formula var)))) formula)
        ((equal? formula var) term)
        ;; the following allows using function calls like (set! x (f y)).
        ((and (list? formula) (equal? (car formula) '=)
              (list? (caddr formula)) (not(list? (cadr formula)))
              (member (caaddr formula) proc-list)
              (not(member (caaddr formula) operator-list)))
         (let ((form
                (get-outputspec
                   (get-prototype (caaddr formula) spec-list))))
           (if (member* (caaddr formula) form)
               (cons '= (subst-h (cdr formula) var term))
               (subst-h
                  (subst (get-outputspec
                           (get-prototype (caaddr formula) spec-list))
                         (append (input-vars (caaddr formula))
                                 (output-vars (caaddr formula)))
                         (append (cdaddr formula)
                                 (list (cadr formula))))
                   var term))))
        ((and (list? formula)
              (member (car formula) proc-list)
              (not(member (car formula) operator-list)))
         (let ((form
                (get-outputspec
                   (get-prototype (car formula) spec-list))))
           (if (member* (car formula) form)
               (cons (car formula)
                     (subst-h (cdr formula) var term))
               (subst-h (subst (get-outputspec
                                 (get-prototype (car formula) spec-list))
                               (append (input-vars (car formula))
                                       (output-vars (car formula)))
                               (cdr formula))
                        var term))))
       (else (cons (subst-h (car formula) var term)
                   (subst-h (cdr formula) var term)))))

(let ((form (simple formula)))
   (if (list? vars)
       (if (and (list? terms) (equal? (length vars) (length terms)))
           (let ((res (subst-h form (car vars) (car terms))))
              (if (empty? (cdr vars)) res
                  (subst res (cdr vars) (cdr terms))))
           (error 'subst: vars 'and terms 'are 'not 'correct
                  'for form))
       (subst-h formula vars terms))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

We conclude this section with the procedures concerning the format check. Note that annotated algorithms' format is not checked once again, because we assume that annotated algorithms are due to the first stage of the generator and hence already have been checked for correct format.

⟨get key of prog 183a⟩ ≡

```
(if (empty? prog)
    (error 'procedure 'get-key: prog 'has 'no 'key)
    (let ((key (get-key prog)))
```

◇

Definition referenced in part 39b.

⟨check format of prog 183b⟩ ≡

```
(check-format prog (get formats key))
```

◇

Definition referenced in part 39b.

"utilities.scm" 183c ≡

```
(define (check-format prog format)

  (define (check-format-h prog format)
   (cond ((and (not(equal? (length prog) (length format)))
               (not(member '* format)))  #f)
         ((or (empty? prog)
              (equal? format '(* *))
              (equal? format '(*))) #t)
         ((equal? (car format) (car prog))
          (check-format-h (cdr prog) (cdr format)))
         ((or (member (car format) symbol-list)
              (member (car prog) proc-list))
          (check-format-h (cdr prog) (cdr format)))
         (else #f) ))

   (cond ((is-annotated? prog) #t)
         ((list? (car prog)) #t)
         ((empty? format)
          (error 'procedure 'check-format:
                 prog 'has 'wrong 'format))
         ((equal? (car format) (get-key prog))
          (if (not (check-format-h prog format))
              (error 'procedure 'check-format:
                     prog 'has 'wrong 'format)))
         (else (if (not (check-format-h prog (car format)))
                   (check-format prog (cdr format)))) ))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

B.2 Table Initialization

In this section we present the necessary initialization of tables. This includes the different kinds of activities as well as some lists of necessary keywords. We start with defining access functions due to the `alist`-package of the `SLib`.

"initialize-tables.scm" 184a ≡

```
(define put (alist-associator equal?))
(define remove (alist-remover equal?))
(define (get table key)
   (let ((result ((alist-inquirer equal?) table key)))
      (if result result
          (error 'procedure 'get: key 'has 'no 'entry))))
(define (insert-newconstruct name format annotation . generation)
  (set! formats (put formats name format))
  (set! annotations (put annotations name annotation))
  (if (not(empty? generation))
      (set! generations (put generations name (car generation))))
  (set! key-list (cons name key-list)))
```

◇

File defined by parts 35c, 184ab, 185ab, 186b.

The following lists include keywords concerning activities representing Hoare rules, logical operations, procedure calls as well as keywords due to the input/output specifications of our example algorithms.[1]

"initialize-tables.scm" 184b ≡

```
(set! key-list
      '(subst proc is-invariant-for is-free?))
(set! logicals-list
      '(true false not and or implies))
(set! symbol-list
      '(condition action action1 action2 var term))
(set! construct-list
      '(inputspec outputspec formalparam actualparam outputparam))
(set! operator-list
      '(+ *))
(set! voc-list
      '(= ~ \in is_normalized_wrt delta < is_associated_to
        divides gcd Amp))
```

◇

File defined by parts 35c, 184ab, 185ab, 186b.

Now rules for annotating algorithms and generating theorems are inserted. Most of these rules have already been presented in chapter three.

"initialize-tables.scm" 185a ≡

```
⟨insert rules 36c, ... ⟩
⟨while rule 54b⟩
⟨assignment rule 53⟩
⟨procedure call rule 54c⟩
⟨return rule 54a⟩
(set! guesses (put guesses 'begin '((rec 'all))))
(set! guesses (put guesses 'return 'none))
```

◇

File defined by parts 35c, 184ab, 185ab, 186b.

The following lists are automatically constructed out of the file `prototypes.txt`, namely a list of the specifications of available subalgorithms, a list of occurring subalgorithm's names and a list of the occurring variable's identifiers.

[1] This keywords could also have been constructed out of the input files; compare for instance the construction of the list of variables below.

"initialize-tables.scm" 185b ≡

```
(set! spec-list '())
(let ((current-input-port (open-input-file "prototypes.txt")))
  (do ((obj (read current-input-port)
            (read current-input-port)))
      ((eof-object? obj) (close-input-port current-input-port))
       (set! spec-list (append spec-list (list obj))) ))
 ⟨make proc-list 185c⟩
 ⟨read variables 186a⟩
```

◇
File defined by parts 35c, 184ab, 185ab, 186b.

⟨make proc-list 185c⟩ ≡

```
(set! proc-list '(g))
(do ((sp-list spec-list (cdr sp-list)))
    ((empty? sp-list) proc-list)
     (set! proc-list
           (cons (oper (car sp-list))
                 proc-list)))
```

◇
Definition referenced in part 185b.

⟨read variables 186a⟩ ≡

```
(set! var-list '())
(do ((specs spec-list (cdr specs)))
    ((empty? specs) #t)
     (let ((lst (append
                   (cdr (get-headline (car specs)))
                   (get-internal-vars (car specs)))))
       (do ((spec lst (cdr spec)))
           ((empty? spec) #t)
            (if (and (not(equal? (car spec) 'out))
                     (not(member (car spec) var-list)))
                (set! var-list (cons (car spec) var-list))) )))
```

◇
Definition referenced in part 185b.

We conclude with the initialization the necessary array for holding the predicates. Note that the natural numbers we introduced in the first stage of the generator when annotating an algorithm serve as the index for the corresponding specific predicate. We use the `array`-package of the SLIB.

"initialize-tables.scm" 186b ≡

```
(require 'array)

(set! predicate-list 'dummy)
(set! get-pred 'dummy)
(set! put-pred 'dummy)


(define (initialize-predlist n)
   (begin
      (set! predicate-list (make-array '* n))
      (set! get-pred
```

```
            (lambda (index)
             (array-ref predicate-list index)))
      (set! put-pred
            (lambda (index object)
              (array-set! predicate-list object index)))
      (do ((nr 0 (+ nr 1)))
          ((= nr n) #t)
          (put-pred nr '?)) ))
```

◇

File defined by parts 35c, 184ab, 185ab, 186b.

B.3 Handling Prototypes

In this section we present access functions for prototypes. They all are easy realized using hardly more than `car` and `cdr`.

"prototypes.scm" 187 ≡

```
(define (get-prototype operator spec-list)
   (cond ((empty? spec-list)
          (error 'Unknown 'operator: operator))
         ((equal? (caadar spec-list) operator)
          (car spec-list))
         (else
          (get-prototype operator (cdr spec-list)))))

(define (get-headline prototype)
   (cadr prototype))

(define (get-whole-input-spec prototype)
   (if (equal? (caaddr prototype) 'internal)
       (cdaddr (cdr prototype))
       (cdaddr prototype)))

(define (get-whole-output-spec prototype)
   (if (equal? (caaddr prototype) 'internal)
       (cdar (cddddr prototype))
       (cdr (cadddr prototype))))

(define (get-internal-vars prototype)
   (if (equal? (caaddr prototype) 'internal)
       (do ((lst (cdaddr prototype) (cdr lst))
            (res '() (append res (list (cadar lst)))))
           ((empty? lst) res))
       '(out)))

(define (get-internals prototype)
   (if (equal? (caaddr prototype) 'internal)
       (caddr prototype)
       (error 'procedure (caadr prototype) 'has 'no 'internals.)))
(define (get-inputspec prototype)
   (let ((input-spec (get-whole-input-spec prototype)))
     (if (equal? (car (last-el input-spec)) 'with)
         (if (> (length (cdr (last-el input-spec))) 1)
```

```
                (cons 'and (cdr (last-el input-spec)))
                (cadr (last-el input-spec)))
            'true)))

    (define (get-outputspec prototype)
       (let ((output-spec (get-whole-output-spec prototype)))
         (if (equal? (car (last-el output-spec)) 'with)
             (if (> (length (cdr (last-el output-spec))) 1)
                 (cons 'and (cdr (last-el output-spec)))
                 (cadr (last-el output-spec)))
             'true)))
```

◇

File defined by parts 187, 188, 189, 190a.

The following procedures allow us to compute the specification of a given algorithm. Note that in absence of the argument `annotated-prog` the result is not the specification of a subalgorithm, but the one of the originally algorithm given by the input file.

"prototypes.scm" 188 ≡

```
    (define (inputspec . annotated-prog)
       (lambda (proc)
          (if (empty? annotated-prog)
              (get-inputspec program-spec)
              (get-inputspec
                 (get-prototype (oper proc) spec-list)))))

    (define (outputspec . annotated-prog)
       (lambda (proc)
          (if (empty? annotated-prog)
              (get-outputspec program-spec)
              (get-outputspec
                (get-prototype (oper proc) spec-list)))))

    (define (internals annotated-prog)
       (get-internals
            (get-prototype (oper annotated-prog) spec-list)))

    (define (head annotated-prog)
       (head-line (oper annotated-prog)))

    (define (inputparam annotated-prog)
       (input-vars (oper annotated-prog)))

    (define (outputparam annotated-prog)
       (output-vars (oper annotated-prog)))
```

◇

File defined by parts 187, 188, 189, 190a.

We conclude this section with some procedures concerning formal and actual parameters of given subalgorithms. Note that the first group of algorithms gets an annotated algorithm as input, whereas the second group only gets an algorithm name.

"prototypes.scm" 189 ≡

```
   (define (actualout proc)
      (lambda (annotated-prog)
         (if (and (list? (cadr annotated-prog))
                  (equal? (caadr annotated-prog) 'call))
             (let ((vars ((actualparam 'proc) annotated-prog))
                   (formout (output-vars (oper annotated-prog))))
                (do ((res vars (cdr res)))
                    ((= (length res) (length formout)) res)))
             (list (cadadr annotated-prog))) ))

   (define (actualparam proc)
      (lambda (annotated-prog)
         (if (and (list? (cadr annotated-prog))
                  (equal? (caadr annotated-prog) 'call))
             (cddadr annotated-prog)
             (append (cdaddr (cadr annotated-prog))
                     (list (cadadr annotated-prog))))))

   (define (formalparam proc)
      (lambda (annotated-prog)
         (append (input-vars (oper annotated-prog))
                 (output-vars (oper annotated-prog)))))
```
◇
File defined by parts 187, 188, 189, 190a.

"prototypes.scm" 190a ≡
```
   (define (headline operator)
      (cadr (get-prototype operator spec-list)))

   (define (input-vars operator)
      (let ((prototype (get-prototype operator spec-list)))
        (do ((var-list (cdadr prototype) (cdr var-list))
             (input-v '() (append input-v (list (car var-list))) ))
            ((equal? (car var-list) 'out) input-v))))

   (define (output-vars operator)
      (let ((prototype (get-prototype operator spec-list)))
      (do ((var-list (cdadr prototype) (cdr var-list)))
          ((equal? (car var-list) 'out) (cdr var-list)))))
```
◇
File defined by parts 187, 188, 189, 190a.

B.4 Input and Output

This section contains SCHEME code due to reading the input files as well as writing into the output files. We only use the standard input and output procedures of SCHEME (see for example [CL91]) using ports.

We start with the SCHEME code for reading the input file. The following assigns the given program to `proglist`.

⟨read input file 190b⟩ ≡

```
(let ((current-input-port (open-input-file inputfile)))
  (let ((obj (read current-input-port)))
    (if (not(is-prototype? obj))
        (set! proglist (append proglist (list obj)))))
  (do ((obj (read current-input-port)
            (read current-input-port)))
      ((eof-object? obj) (close-input-port current-input-port))
       (set! proglist (append proglist (list obj))) ))
```

◇
Definition referenced in parts 48b, 52a, 56ac.

The next piece of code handles the prototype given by the input file: First, the specification is assigned to `program-spec`. Then the subalgorithm list (`proc-list`), the specification list (`spec-list`) and the list of variables (`var-list`) are updated.

⟨read program specs 190c⟩ ≡

```
(let ((current-input-port (open-input-file inputfile)))
(set! program-spec (read current-input-port))
(set! proc-list (cons (oper program-spec) proc-list))
(set! spec-list (cons program-spec spec-list))
(let ((vars (append (cdr (get-headline program-spec))
                    (get-internal-vars program-spec))))
   (do ((var vars (cdr var)))
           ((empty? var) #t)
            (if (and (not(equal? (car var) 'out))
                     (not(member (car var) var-list)))
                (set! var-list (cons (car var) var-list))) ))
(close-input-port current-input-port))
```

◇
Definition referenced in part 48b.

The rest of this section contains the SCHEME code concerning writing into the output files.

⟨open output file 191a⟩ ≡

```
(let ((current-output-port (open-output-file outputfile)))
```

◇
Definition referenced in parts 49a, 52a, 56c, 192b.

⟨write block 191b⟩ ≡

```
(write block current-output-port)
```

◇
Definition referenced in part 49b.

⟨close output file 191c⟩ ≡

```
(close-output-port current-output-port)
```

◇
Definition referenced in parts 49b, 56c, 192b.

The following writes the constructed abstract theorems into the output file. Note that side conditions are not checked for correctness, but only listed in the output file.

⟨write+close output file 192a⟩ ≡

```
(do ((theorems (reverse theorem-list) (cdr theorems)))
    ((empty? theorems) )
     (begin (write (car theorems) current-output-port)
            (newline current-output-port)
            (newline current-output-port)))
(newline current-output-port)
(newline current-output-port)
(newline current-output-port)
(do ((side-conds (reverse side-cond-list) (cdr side-conds)))
    ((empty? side-conds) (close-output-port current-output-port))
     (begin (write (car side-conds) current-output-port)
            (newline current-output-port)
            (newline current-output-port)))
```

◇

Definition referenced in part 52b.

During writing the constructed specific theorems into the output file, we do two further things: We simplify the given theorems using procedure `simple` and we check whether the computed side conditions hold; this concerns theorems starting with the phrase `'is-invariant-for`.

⟨write theorems 192b⟩ ≡

```
⟨open output file 191a⟩
(do ((nr 0 (+ nr 1)))
    ((= nr (+ prednr 1)) (newline current-output-port))
     (begin
        (write (get-pred nr) current-output-port)
        (newline current-output-port)
        (newline current-output-port)))
(newline current-output-port)
(newline current-output-port)
(do ((theorems proglist (cdr theorems)))
    ((empty? theorems) ⟨close output file 191c⟩ )
     (let ((form (expand (car theorems))))
        (cond ((equal? (caar theorems) 'implies)
               (begin
                  (if (equal? form '?)
                      (write (car theorems) current-output-port)
                      (write (simple (do-subst form))
                             current-output-port))
                  (newline current-output-port)
                  (newline current-output-port)))
              ((equal? (caar theorems) 'is-invariant-for)
               (begin
                  (if (equal? form '?)
                      (write (car theorems) current-output-port)
                      (if (not(is-invariant?
                                (cadr form) (caddr form)))
                          (write (list 'not form '!)
                                 current-output-port)))
                  (newline current-output-port)
                  (newline current-output-port)))

              (else
               (error 'unknown 'kind 'of 'theorem:
```

```
                          (car form))) ))) )
```
◇
Definition referenced in part 56a.

We also compute a `missing-list`: It holds the numbers of the abstract predicates left without a specific counterpart.

⟨guesses message 193a⟩ ≡
```
     (set! missing-list '())
     (do ((nr 0 (+ nr 1)))
         ((equal? nr prednr) (if (not(empty? missing-list))
                                 (begin
                                   (display "predicate(s) ")
                                   (write (reverse missing-list))
                                   (display " fail!") (newline))))
          (if (equal? (get-pred nr) '?)
              (set! missing-list (cons nr missing-list)) ))
```
◇
Definition referenced in part 56b.

The following piece of code is to read over the computed predicates during the call of `make-trivial-theorems`.

⟨handle predicates 193b⟩ ≡
```
     (do ((nr 0 (+ nr 1)))
         ((= nr (+ prednr 1)) (newline current-output-port))
          (set! proglist (cdr proglist)))
```
◇
Definition referenced in part 56c.

B.5 Additional Functions

We conclude with some technical procedures completing our verification condition generator. Note again that the use of `StWeb` allows extracting the source code of the generator out of this document.

"utilities.scm" 194a ≡
```
    (define (empty? prog)
       (null? prog))

    (define (oper obj)
       (cond ((is-annotated? obj)
              (if (and (list? (cadr obj))
                       (equal? (caadr obj) 'call))
                  (cadadr obj)
                  (caaddr (cadr obj))))
             ((is-prototype? obj) (caadr obj))
             (else (error 'procedure 'oper: obj 'is 'unknown))))


    (define (is-prototype? obj)
       (and (list? obj) (equal? (car obj) 'prototype)))
```

```
   (define (is-annotated? prog)
      (or (and (number? (car prog)) (not(empty? (cdr prog))))
          (and (list? (car prog))
               (member (caar prog) logicals-list))))

   (define (is-not-already-specific obj)
      (and (number? obj) (equal? (get-pred obj) '?)))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

"utilities.scm" 194b ≡

```
   (define (get-key prog)
      (if (is-annotated? prog)
          (get-key (cadr prog))
          (if (list? (car prog))
              (caar prog) (car prog))))

   (define (is-prednr? obj)
      (or (number? obj) (equal? obj '?)))

   (define (without-last lst)
      (cond ((or (not(list? lst)) (empty? lst))
             (error 'procedure 'cdr-without-last 'needs 'nonempty 'lst))
            ((empty? (cddr lst)) (list (car lst)))
            (else (cons (car lst) (without-last (cdr lst)) )) ))

   (define (last-el lst)
      (cond ((empty? lst)
             (error 'procedure 'last-el 'needs 'non 'empty 'list))
            ((empty? (cdr lst)) (car lst))
            (else (last-el (cdr lst)))))

   (define (member* obj list)
      (cond ((or (empty? list)
                 (and (not(list? list))
                      (not(equal? obj list)))) #f)
            ((equal? obj list) #t)
            ((member obj list) #t)
            (else (or (member* obj (car list))
                      (member* obj (cdr list))))))

   (define (is-sequence-without-begin? obj)
      (and (list? obj) (not(member (car obj) logicals-list))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

"utilities.scm" 195 ≡

```
   (define (fits? prog format)
      (if (or (equal? (get-key prog) 'if)
              (equal? (get-key prog) 'return))


          (if (is-annotated? prog)
              (equal? (length (cadr prog)) (length format))
```

```
            (equal? (length prog) (length format)))
        #t))

   (define (actual? activity prog)
      (if (equal? (get-key prog) 'if)
          (if (is-annotated? prog)
              (if (or (> (length (cadr prog)) 3)
                      (equal? (car activity) 'set-predicate)) #t
                  (equal? (cadr (cadadr activity)) 'action1))
              (if (> (length prog) 3) #t
                  (equal? (eval (cadr activity)) 'action1)))
          #t))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

We carry on with two further procedures concerning the check whether a variable is free in a given object.

"utilities.scm" 196a ≡

```
   (define (is-not-free obj formula)
      (not (is-free obj formula)))

   (define (is-included obj annotated-prog)
      (or (member (list obj) (caddr (cadr annotated-prog)))
          (do ((lst (caddr (cadr annotated-prog)) (cdr lst)))
              ((or (empty? lst)
                   (and (list? (car lst))
                        (equal? (caar lst) obj)))
               (not(empty? lst))))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

What follows are the procedures that realize the test whether a given theorem is trivial. We included tests for *implication*, *negation* and *equality*.

"utilities.scm" 196b ≡

```
    (define (is-trivial theorem)
        (cond ((equal? theorem 'true) #t)
              ((equal? theorem 'false) #f)
              ((not(list? theorem)) #f)
              (else
               (let ((key (get-key theorem)))
                 (cond
                   ((equal? key 'implies) (is-trivial-imply? theorem))
                   ((equal? key '=) (equal? (cadr theorem) (caddr theorem)))
                   ((equal? key 'and) (is-trivial-and? theorem))
                   (else #f))) )))

    (define (is-trivial-and? theorem)
       (if (empty? theorem) #t

           (do ((args (cdr theorem) (cdr args))
                (res #t (if (is-trivial (car args)) res #f)))
```

```
            ((empty? args) res))))

(define (is-trivial-imply? theorem)
   (let ((ass (cadr theorem))
         (concl (caddr theorem)))
      (cond ((or (empty? concl) (equal? ass concl)
                 (equal? ass 'false) (equal? concl 'true)) #t)
            ((not(list? concl))
             (or (equal? ass concl) (is-trivial concl)
                 (and (list? ass) (member concl ass))))
            ((and (list? concl)
                  (or (member (car concl) logicals-list)
                      (member (car concl) voc-list)
                      (member (car concl) operator-list)))
             (if (equal? (car concl) 'and)
                 (if (empty? (cdr concl)) #t
                     (let ((res (is-trivial-imply?
                                 (list 'implies ass (cadr concl)))))
                       (if res
                           (is-trivial-imply?
                            (list 'implies ass (cons 'and (cddr concl))))
                           #f)))
                 (or (equal? ass concl) (is-trivial concl)
                     (and (list? ass) (member concl ass)))))
            (else
             (and (is-trivial-imply?
                     (list 'implies ass (car concl)))
                  (is-trivial-imply?
                     (list 'implies ass (cdr concl))))) )))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

Procedure `simple` transforms a constructed theorem into a more readable form. To give an example, `(not(not A))` is replaced by `A`.

"utilities.scm" 197 ≡

```
(define (simple theorem)
   (if (not(list? theorem))  theorem
       (let ((key (get-key theorem)))
         (cond ((equal? key 'not)
                (if (and (list? (cadr theorem))
                         (equal? (caadr theorem) 'not))
                    (simple (cadadr theorem))
                    (list 'not (simple (cadr theorem)))))
               ((equal? key 'implies)
                (cond ((equal? (simple (cadr theorem)) 'true)
                       (simple (caddr theorem)))
                      ((equal? (simple (cadr theorem)) 'false)
                       'true)
                      ((equal? (simple (caddr theorem)) 'true)
                       'true)

                      ((equal? (simple (caddr theorem)) 'false)
                       (simple (list 'not (cadr theorem))))
```

```
                    (else
                     (list 'implies (simple (cadr theorem))
                           (simple (caddr theorem))))))
               ((equal? key 'and)
                (do ((args (cdr theorem) (cdr args))
                     (res (list 'and)
                          (let ((arg (simple (car args))))
                             (cond ((equal? arg 'true) res)
                                   ((equal? arg 'false)
                                    (list 'false))
                                   ((and (list? arg)
                                         (equal? (get-key arg) 'and))
                                    (append res (cdr arg)))
                                   (else (append res (list arg))))))
                    ((empty? args) (if (member 'false res) 'false
                                       (if (= (length res) 2)
                                           (cadr res)
                                           res))))
                (else theorem)))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

Procedure `expand` gets a formula as input. This formula may still contain abstract predicates. The procedure replaces these predicates by using the already mentioned array that holds the computed conjectures for the abstract predicates.

"utilities.scm" 198 ≡

```
(define (expand formula)

   (define (expand-h formula)
      (cond ((or (empty? formula) (member formula key-list)
                 (member formula logicals-list) (member formula voc-list)
                 (member formula proc-list) (member formula var-list))
             formula)
            ((and (list? formula)
                  (member (car formula) operator-list))
             (list (car formula)
                   (expand-h (cadr formula))
                   (expand-h (caddr formula))))
            ((and (list? formula)
                  (equal? (car formula) '=)
                  (not(list? (caddr formula)))) formula)
            ;; the following allows function calls like (set! x (f y)).
            ((and (list? formula)
                  (equal? (car formula) '=)
                  (not(list? (cadr formula)))
                  (list? (caddr formula))
                  (member (caaddr formula) proc-list)
                  (not(member (caaddr formula) operator-list)))



             (subst (get-outputspec
                      (get-prototype (caaddr formula) spec-list))
```

```
                (append (input-vars (caaddr formula))
                        (output-vars (caaddr formula)))
                (append (cdaddr formula)
                        (list (cadr formula)))))
          ((and (list? formula) (equal? (car formula) 'subst))
           (cons 'subst
                 (cons (expand-h (cadr formula))
                       (cddr formula))))
          ((is-prednr? formula)
           (if (equal? (get-pred formula) '?) '?
               (expand-h (get-pred formula))))
          ((list? formula)
           (cons (expand-h (car formula))
                 (expand-h (cdr formula))))
          (else (error 'procedure 'expand: formula 'is 'unknown))))

  (let ((form (expand-h formula)))
    (if (member* '? form) '?
        (simple form))))
```

◇

File defined by parts 179, 180ab, 181, 183c, 194ab, 195, 196ab, 197, 198.

Appendix C

Indices

C.1 Files

C.2 Macros

⟨ann-rec all 47a⟩ Referenced in part 46b.
⟨ann-rec special 47b⟩ Referenced in part 46b.
⟨annotate main loop 49b⟩ Referenced in part 49a.
⟨assignment rule 53⟩ Referenced in part 185a.
⟨cases of theorem L11, if 22a⟩ Referenced in part 21b.
⟨check format of prog 183b⟩ Referenced in part 39b.
⟨check other formats 48a⟩ Referenced in part 47b.
⟨close output file 191c⟩ Referenced in parts 49b, 56c, 192b.
⟨correctness proof of Classes 137⟩ Referenced in part 60b.
⟨correctness proof of Class 135⟩ Referenced in part 60b.
⟨correctness proof of gcd function 145b⟩ Referenced in part 24.
⟨correctness proof of normal form 144⟩ Referenced in part 71a.
⟨environment 15b⟩ Referenced in part 15a.
⟨example lemma 20c⟩ Referenced in part 124.
⟨existence proof for Euclidean domains 95ab⟩ Referenced in part 94.
⟨existence proof of AmpSet 61, 62abc, 63ab⟩ Referenced in part 60a.
⟨existence proof of AmpleSet 139⟩ Referenced in part 69b.
⟨existence proof of fractions 76b⟩ Referenced in part 76a.
⟨existence proof of gcdDomain 72a⟩ Referenced in part 71c.
⟨function call 50b⟩ Referenced in part 50a.
⟨gcd theorems 25⟩ Referenced in part 146.
⟨gen-rec all 51a⟩ Referenced in part 50c.
⟨gen-rec special 51b⟩ Referenced in part 50c.
⟨get key of prog 183a⟩ Referenced in part 39b.
⟨guesses message 193a⟩ Referenced in part 56b.
⟨handle predicates 193b⟩ Referenced in part 56c.
⟨insert rules 36c, 38b, 39a, 41a⟩ Referenced in part 185a.
⟨insert-pred-after all 45b⟩ Referenced in part 45a.
⟨insert-pred-after special 46a⟩ Referenced in part 45a.
⟨lemma for Euclidean algorithm 133⟩ Referenced in part 92b.
⟨lemmata for Brown/Henrici 131⟩ Referenced in part 164.
⟨make proc-list 185c⟩ Referenced in part 185b.
⟨make-guesses main loop 56b⟩ Referenced in part 56a.
⟨make-theorems main loop 52b⟩ Referenced in part 52a.
⟨open output file 191a⟩ Referenced in parts 49a, 52a, 56c, 192b.
⟨procedure call generations 38a⟩ Referenced in part 38b.
⟨procedure call rule 54c⟩ Referenced in part 185a.
⟨proof of AMP5 142b⟩ Referenced in part 70b.
⟨proof of B11 109a⟩ Referenced in part 106b.
⟨proof of B1 111b⟩ Referenced in part 105a.
⟨proof of B63 111c⟩ Referenced in part 109c.
⟨proof of B6 109bc⟩ Referenced in part 109a.
⟨proof of B7 110ab, 111a⟩ Referenced in part 109a.
⟨proof of Brown/Henrici theorem 27b, 28abc, 29a⟩ Referenced in part 23b.
⟨proof of F1 157a⟩ Referenced in part 80a.
⟨proof of F2 157b⟩ Referenced in part 81b.
⟨proof of H0 68c⟩ Referenced in part 68b.
⟨proof of H11 29b⟩ Referenced in part 28b.
⟨proof of H14 30b⟩ Referenced in part 28c.
⟨proof of H1 69a⟩ Referenced in part 68b.
⟨proof of H2b 68a⟩ Referenced in part 66a.
⟨proof of H2 100a⟩ Referenced in part 99b.
⟨proof of H3 100b⟩ Referenced in part 99b.
⟨proof of H5 112⟩ Referenced in part 111c.

⟨proof of H7 30a⟩ Referenced in part 28a.
⟨proof of K2 65a⟩ Referenced in part 62a.
⟨proof of K3 65b⟩ Referenced in part 62a.
⟨proof of K5a 65c⟩ Referenced in part 62b.
⟨proof of K6a 66abc, 67a⟩ Referenced in part 63a.
⟨proof of K6 68b⟩ Referenced in part 63a.
⟨proof of K7 63c⟩ Referenced in part 63b.
⟨proof of K8 64ab⟩ Referenced in part 63b.
⟨proof of M4 67b⟩ Referenced in part 66a.
⟨proof of N 104a⟩ Referenced in part 103b.
⟨proof of case A 104b⟩ Referenced in part 104a.
⟨proof of case B 105ab, 106abc, 108abc⟩ Referenced in part 104a.
⟨proof of denom 156c⟩ Referenced in part 79b.
⟨proof of fraction addition 158a⟩ Referenced in part 81a.
⟨proof of fraction's constructor equation 156b⟩ Referenced in part 79a.
⟨proof of gcd-like, case A, label A5 73c, 74⟩ Referenced in part 73b.
⟨proof of gcd-like, case A 73b⟩ Referenced in part 72b.
⟨proof of gcd-like, case B 73a⟩ Referenced in part 72b.
⟨proof of gcd-like 72b⟩ Referenced in part 72a.
⟨proof of theorem L11, if, case A 22b⟩ Referenced in part 22a.
⟨proof of theorem L11, if, case B 23a⟩ Referenced in part 22a.
⟨proof of theorem L11, if 21b⟩ Referenced in part 20d.
⟨proof of theorem L11, only if 21a⟩ Referenced in part 20d.
⟨proof of theorem L11 20d⟩ Referenced in part 20c.
⟨proof of theorem T0 151a⟩ Referenced in part 25.
⟨proof of theorem T1 151b⟩ Referenced in part 25.
⟨proof of theorem T2 153⟩ Referenced in part 25.
⟨proof of theorem T3 26abc, 27a⟩ Referenced in part 25.
⟨proof of theorem T4 154⟩ Referenced in part 25.
⟨read input file 190b⟩ Referenced in parts 48b, 52a, 56ac.
⟨read program specs 190c⟩ Referenced in part 48b.
⟨read variables 186a⟩ Referenced in part 185b.
⟨return rule 54a⟩ Referenced in part 185a.
⟨text proper 16, 123⟩ Referenced in part 15a.
⟨while annotations 36a⟩ Referenced in part 36c.
⟨while generations 36b⟩ Referenced in part 36c.
⟨while rule 54b⟩ Referenced in part 185a.
⟨write block 191b⟩ Referenced in part 49b.
⟨write theorems 192b⟩ Referenced in part 56a.
⟨write+close output file 192a⟩ Referenced in part 52b.

C.3 Procedure Names