A Constructive Formalisation of Semi-algebraic Sets and Functions

Boris Djalal

▶ To cite this version:

Boris Djalal. A Constructive Formalisation of Semi-algebraic Sets and Functions. CPP 2018 - Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2018, Los Angeles, California, United States. pp.240-251. hal-01643919

HAL Id: hal-01643919

https://inria.hal.science/hal-01643919

Submitted on 21 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Constructive Formalisation of Semi-Algebraic Sets and Functions

Boris Djalal
Inria Sophia Antipolis – Méditerranée, France
France
boris.djalal@gmail.com

Abstract

Semi-algebraic sets and semi-algebraic functions are essential to specify and certify cylindrical algebraic decomposition algorithms. We formally define in Coq the base operations on semi-algebraic sets and functions using embedded first-order formulae over the language of real closed fields, and we prove the correctness of their geometrical interpretation. In doing so, we exploit a previous formalisation of quantifier elimination on such embedded formulae to guarantee the decidability of several first-order properties and keep our development constructive. We also exploit it to formalise formulae substitution without having to handle bound variables.

Keywords Formalisation of Mathematics, Semi-Algebraic Sets, Semi-Algebraic Functions, Coq, Quantifier Elimination, Real Algebraic Geometry, Substitution

1 Introduction

First-order formulae over real closed fields, which can express a wide range of problems (polynomial optimisation, topologically reliable algebraic curve display, termination proof of term-rewriting systems) [5] are decidable. Quantifier elimination, that consists in finding a logically equivalent formula without quantifiers, is the keystone of the decision procedures for first-order formulae presented in the literature [5]. The first quantifier elimination algorithm [24] has complexity a tower of exponents of height linear in the number of variables [2]. The cylindrical algebraic decomposition (abbreviated by CAD from now on) was invented by George E. Collins to eliminate quantifiers with a better complexity than Tarski's original algorithm [1, 3]: the complexity reduces to a double exponential in the number of quantifiers. The expression "CAD" denotes two different notions: the algorithm and its output. In this paper, we call "CAD" a partition of the geometric space into semi-algebraic (abbreviated by *S.A.* from now on) sets, which satisfies some additional properties as explained in Sect. 4.1. We call "CAD algorithm" any algorithm that returns such a partition. Moreover, CAD algorithms help to answer questions about central objects in real algebraic geometry, *S.A.* sets. For example: how to compute sample points of a given nonempty *S.A.* set or the number of points of a *S.A.* set, decide whether a *S.A.* set is open, closed or bounded, determine the connected components of a *S.A.* set [2].

To work on the correctness proof of a given CAD algorithm, one firstly needs to tackle the two following problems: formalise what constitutes a CAD output, then formalise what constitutes a CAD algorithm.

In the present work, we formalise two key concepts required to formalise what constitutes the output of a CAD, *S.A.* sets and *S.A.* functions, while exploiting a previous quantifier elimination formalised by Cohen and Mahboubi [9]. Defining *S.A.* sets and *S.A.* functions brings us to solve intermediary problems to represent such objects in Coq.

In Sect. 2, we introduce preliminary notions required by the CAD, by *S.A.* sets and by *S.A.* functions.

Firstly, in Sect. 2.1, we briefly clarify what is a first order formula on a real closed field. On top of first-order formulae, we define sets of free variables in Sect. 2.2. This enables us to build the type of formulae with free variables in $\{X_0, \cdots, X_{n-1}\}$ in Sect. 2.4. We remind what is quantifier elimination and how we use the one from Mathematical Components [21] in Sect. 2.3.

In particular, we use it in Sect. 3 to define the logical equivalence relation on such formulae in a decidable way (to keep our development constructive).

We describe the construction of the type of *S.A.* sets in Sect. 4. Firstly, we define the CAD and explain why we need the *S.A.* set concept in Sect. 4.1. Secondly, we formalise *S.A.* sets in Sect. 4.2 through a quotient structure by combining results on formulae. We prove that two *S.A.* sets are equal if, and only if, they contain the same elements. We then equip *S.A.* sets with a lattice structure.

In Sect. 5, we describe our formalisation of *S.A.* functions, whose graphs are *S.A.* sets. This formalisation requires to express functionality and totality of *S.A.* graphs in a decidable way (to keep our development constructive). We achieve this by constructing tailor-made reified formulae expressing functionality and totality and by proving their correctness, similarly to the reification technique presented in [16, 20, 23]. These constructions rely on the substitution in formulae. We do not define the substitution in the usual way, as in [13, 14]. Instead, we simplify the definition and specification of the substitution in formulae by exploiting quantifier elimination, in Sect. 6.1.

CPP'18, January 07–08, 2018, Los Angeles, LA, USA
2018.

Using the same methodology as for functionality and totality, we show how to formalise the composition of two *S.A.* functions in Sect. 7 and the continuity of a *S.A.* function in Sect. 8. The latter turns out to be more difficult and we provide an incomplete proof of correctness.

In the present paper, we present modified snippets of our code available at https://github.com/math-comp/cad.

2 Preliminary Notions

In this section, we briefly remind (2.1) what is quantifier elimination (2.3) in the Mathematical Components settings; we present our contributions about free variables (2.2) and formulae (2.4) (3).

2.1 First-Order Formulae over Real Closed Fields

In this paper, we use first-order formulae over real closed fields to formalise *S.A.* sets. We now clarify these two notions.

One definition is that a real closed field is an ordered field that has no ordered algebraic extension [7]. In particular, it is equipped with the arithmetic operations (addition, negation, multiplication and thus exponentiation with a natural number) and with comparison. For example, the real numbers and the real algebraic number are real closed fields. To keep the analogy with real numbers, we denote a real closed field by $\mathbb{R}$. However, in all our work, we do no suppose that $\mathbb{R}$ is archimedean, whereas the real numbers is archimedean.

We use operations on $\mathbb{R}$ to define terms and first-order formulae. A term over $\mathbb{R}$ is a variable X_i, $i \in \mathbb{N}$ or else a formal operation over terms. Formal arithmetic operations are: addition, multiplication, exponentiation, and negation. A formula over a field $\mathbb{R}$ is defined inductively as a formal comparison of two terms or else as a formal logical operation on formulae. A formal logical operation is either a formal logical connective (and, or, implication, negation) or a formal quantification of a formula over a variable — quantifying over an already bounded variable or missing variable is thus allowed. For example, $\exists X_0 \exists X_0 X_0 = 0$, $\exists X_0 X_1 = 0$ and $\exists X_2 \exists X_3 \Big((X_2 + X_3 - X_0 = 0) \wedge (\frac{1}{3} X_2 X_3 - X_1) \wedge (X_2 < X_3)\Big)$ are valid formulae over $\mathbb{R}$.

In our work, we exploit a previous formalisation of real closed fields and first-order formulae from Mathematical Components.

2.2 Free Variables

The free variables of a first-order formula are the variables that are not bound by a quantifier. They are the dimensions of $\mathbb{R}^n$ that matter in the geometric interpretation of the formula, so they play a central role in *S.A.* sets over $\mathbb{R}^n$ as described in Sect. 2.4. We compute the free variables recursively as follows: the free variables of the term X_i is the set $\{X_i\}$; the free variables of a quantification formula of the form $\mho X_i\, \phi$, $\mho \in \{\forall, \exists\}$, are the free variables of the subformula ϕ minus X_i. In all other cases, the free variables of a formula is the union of the free variables of its subformulae. For example, the free variables of $\exists X_2 \exists X_3 \Big((X_2 + X_3 - X_0 = 0) \wedge (\frac{1}{3} X_2 X_3 - X_1) \wedge (X_2 < X_3)\Big)$ is the set $\{X_0\}$.

In Coq, finite sets can be represented by sequences without duplicates. With this representation, In our formalisation, we use the appropriate, higher level, notion of finite set from Mathematical Components instead of sequences. It allows us to work with the Coq equality (`=`) on finite sets, instead of the extensional equivalence (`=i` in Coq) on sequences of variables. Moreover, we automatically get a decidable equality on sets of variables, because the equality on natural numbers is decidable. We use this to keep our development constructive (but we do not run any decision procedure).

2.3 Semantics of First-Order Formulae and Quantifier Elimination

Terms and formulae are geometrically interpreted respectively as elements of $\mathbb{R}$ and Coq first-order formulae. The interpretation has already been defined by Cohen [9] and corresponds to two functions `eval` and `holds`, which take an environment e to give values to free variables and respectively output an element of $\mathbb{R}$ and an element of `Prop`. An open formula with free variables in $X_0, \cdots, X_{n-1}$ can be viewed as a predicate on $\mathbb{R}^n$, i.e. a subset of $\mathbb{R}^n$.

In this work we exploit the formalisation of a quantifier elimination procedure from a previous work by Cohen and Mahboubi [9], which applies to formulae over a real closed field.

Firstly we use the existing decision procedure, `rcf_sat`, to decide whether a given first-order `formula f` is true in a given environment `e`. It is expressed b ythe expression `rcf_sat e f` (with type `bool`) in Coq where `e` assignes values to all free variables of `f`. When `f` is closed, this is written as `rcf_sat [::] f`. In particular, we use this decision procedure to define the decidable logical equivalence of two first-order formulae in Sect. 3.

Secondly, we use the quantifier elimination procedure in Sect. 6.1. A quantifier elimination procedure returns an equivalent quantifier-free formula. However, the existing quantifier elimination procedure, `quantifier_elim`, is not specified enough to guarantee that no free variable is introduced. Based on `quantifier_elim`, we define another one, `qf_elim`, which does not introduce new free variable, as follows. In Coq, we consider an input formula `f` and the quantifier-free equivalent `g` given by `quantifier_elim`. The formula `g` may introduce new free variables from the variables set ``formula_fv g `\` formula_fv f``. The instantiation of these extra variables does not affect `f`. Thus, the instantiation of these extra variables in `g` returns a formula equivalent to `g`. We choose to instantiate these variables in `g` with 0. We define `quantifier_elim` in Coq. We formally prove that the resulting formula is equivalent to `f`, that it

221 is quantifier-free, and the following statement that its free variables are among those of f:

```
Lemma qf_elim_fv (f : formula R) :
formula_fv (qf_elim f) <= formula_fv f.
```

where <= denotes the subset relation.

2.4 Formulae with less than n Free Variables

S.A. sets of $\mathbb{R}^n$ involve a finite number of variables among $X_0, \cdots, X_{n-1}$. Since we can find an equivalent formula without bound variables (Sect. 2.3), we consider formulae with free variables among $X_0, \cdots, X_{n-1}$, which we denote by $\mathscr{F}_n = \{\varphi \in \mathscr{F} \mid \text{freevar}(\varphi) \subseteq \{X_0, \cdots, X_{n-1}\}\}$. In Coq, we denote $\mathbb{R}$ by R and n by n and we encode $X_0, \cdots, X_{n-1}$ by the variables 'X_0, ..., 'X_(n - 1) and we define the decidable predicate $\text{freevar}(\varphi) \subseteq \{X_0, \cdots, X_{n-1}\}$ by:

```
Definition nvar (n : nat) :=
fun (f : formula R) => formula_fv f <= mnfset 0 n.
```

where mnfset 0 n is the set of natural numbers ranging from 0 to $n - 1$. The set $\mathscr{F}_n$ is then formalised by:

```
Record formulan := MkFormulan
{
  underlying_f : formula R ;
  _ : nvar underlying_f
}.
Notation "'{formula_' n R }" := (formulan R n).
```

Formulae over $\mathbb{R}$ with less than n free variables inherit the choice type structure from $\mathscr{F}$. Given any inhabitated predicate (inhabitated decidable property) over a choice type structure, one can choose a witness of that predicate. We choose the same witness for any logically equivalent property. See the Mathematical Components book [21] for more information on choice types.

We automatically cast elements of $\mathscr{F}_n$ to elements of $\mathscr{F}$ using the Coq implicit coercion mechanism:

```
Coercion underlying_f : formulan >-> formula.
```

The theorems which apply to $\mathscr{F}$ also apply to $\mathscr{F}_n$, by coercion. For example, the formula constructor And expects two elements of $\mathscr{F}$, yet we can apply it to two elements f and g of $\mathscr{F}_n$ and get the element f /\ g of $\mathscr{F}$.

In our example, f /\ g still has n variables. We prove this in Coq:

```
Lemma and_formulan (f g : {formula_n R}) : nvar n (f /\ g)%oT.
```

where oT makes interpret the symbol /\ in the scope of formulae. This situation is similar to tuples in Mathematical Components. Tuples are sequences with a given length. When we concatenate two tuples, we obtain a sequence whose length is the sum of the lengths. In Mathematical Components, this piece of information is recovered automatically by declaring a canonical solution [21].

In a similar way, we automatically recover the piece of information about f /\ g by declaring and_formulan as a canonical solution:

```
Canonical Structure formulan_and f g :=
  MkFormulan (and_formulan f g).
```

Then, the system is able to build the proof of nvar underlying_f on the fly for any two specific values f and g, and lift f /\ g to $\mathscr{F}_n$. We implement similar solutions for other operations.

2.5 Decidable Equivalence Relation over Formulae with less than n Free Variables

To define *S.A.* sets in (see Sect. 4), we need to define the logical equivalence relation over $\mathscr{F}_n$.

In all this Sect., we suppose that we have an equivalence relation equivf over $\mathscr{F}$ (see Sect. 3) such that its restriction to $\mathscr{F}_n$ is the logical equivalence relation over $\mathscr{F}_n$. We show how to formalise this restriction in a more general settings.

Since we can use elements of $\mathscr{F}_n$ in place for elements of $\mathscr{F}$, we can see equivf as an equivalence relation on $\mathscr{F}_n$. In Coq, we define a new relation on {formula_n R} by restricting equivf to formulae with free variables in $X_0, \cdots, X_{n-1}$; we then prove that this restriction is an equivalence relation.

Instead of directly proving that the subrelation induced by equivf on formula_n F is also an equivalence relation, we generalise this construction over the types formula and formula_n R: any equivalence relation on a type T induces an equivalence relation sub_r on any subtype of T (subtype structures are explained in the Mathematical Components book [21]). We define sub_r in Coq by:

```
Variables (T : eqType) (P : pred T) (sT : subType P) (r : equiv_rel T).
Definition sub_r (x y : sT) := r (val x) (val y).
```

We bring down reflexivity (sub_r_refl), symmetry (sub_r_sym) and transitivity (sub_r_trans) of sub_r to the reflexivity, symmetry and transitivity of equiv_rel, respectively. This way, we formally prove these three properties for the subrelation sub_r. We make the equivalence structure of the subrelation a canonical structure, so that this structure is recovered automatically by the system:

```
Canonical sub_r_equiv :=
EquivRel sub_r sub_r_refl sub_r_sym sub_r_trans.
```

Our generic canonical solution takes the proof that the broader relation is an equivalence relation and the desired subtype, then automatically builds the equivalence relation induced on the subtype.
We apply this generic construction to define the logical equivalence sub_equivf on $\mathscr{F}_n$, in Coq:

```
Definition sub_equivf :=
 @sub_r _ _ [subType of {formula_n R}] equivf_equiv.
```

331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385

3 First Reification case for the Equivalence Relation over Formulae

In this section, we define a logical equivalence relation on formulae, such that its restriction to $\mathscr{F}_n$ (see Sect. 2.5) is a decidable equivalence. We show how to use reification to express this relation in a decidable way.

Let ϕ and ψ denote two formulae. We consider the following relation: ϕ and ψ are equivalent if they evaluate to the same truth-value in all environments of size n (any free variable X_i with $n < i$ is assigned to the default value 0). In Coq, this can be expressed by the following property of type `Prop`: `forall (e : n.-tuple F), holds e f <-> holds e g`, where `e` is a sequence of size `n` (in other words `e` is a tuple of size `n`, denoted `n.-tuple F`). To write a decidable version of this property, we first need to express it through the specific formula type expected by `rcf_sat` (this is called reification), then evaluate its boolean truth-value with `rcf_sat`. We firstly remark that `holds e f <-> holds e g` is the definition of `holds e (f <==> g)` where `<==>` is the equivalence in the language of real closed fields. We are thus able to express the connector `<->` on `Prop` in the language of real closed fields. (In our code, the constant `<==>` is not directly part of the language, but we define it in terms of `==>` and `/\`.)

To complete the reification, we push quantification over `e` inward, so that quantification over `e` is replaced by n quantifications over variables. The latter quantifications are part of the targeted first-order language. A reified version of the property above thus has the form $\forall X_0 \cdots \forall X_{n-1}\, \phi \iff \psi$, in Coq:

```
nquantify 0 n Forall (f <==> g)
```

Listing 1.

where the function `nquantify i n` Ɑ prefixes any formula with the block quantification $\mho X_i \cdots \mho X_{i+n-1}$ from the index i up to the index $i+n-1$ with Ɑ being one of the quantifiers $\forall$ or $\exists$. We illustrate the working of `nquantify` on the following example. The application of `nquantify 3 2 Forall` on the formula $\exists X_5\,(X_5 = X_3 \lor X_5 = X_4) \land X_3 \le X_5 \land X_4 \le X_5$ outputs the formula $\forall X_3 \forall X_4 \exists X_5\,(X_5 = X_3 \lor X_5 = X_4) \land X_3 \le X_5 \land X_4 \le X_5$. The resulting formula is closed (we remark that it means that any numbers pair (X_3, X_4) have a maximum X_5). This formula (Listing 1) is called a reified formula according to the reification techniques presented in other works [16, 20, 23]. Since it is a first-order formula in the expected type, we can apply `rcf_sat` on it. We evaluate it in the empty environment (which assigns any free variable to the default assignment value 0). In Coq, a decidable version of our equivalence relation is expressed by:

```
Definition equivf (f g : formula R) :=
rcf_sat [::] (nquantify 0 n Forall (f <==> g)).
```

Our decidable version is used for the only sake of constructivity of our development, we do not use it to make computations; contrary to other works [16, 20, 23], where the reification is used to compute decision procedures to replace proof steps with computations (reflection). The second point of comparison with these works [16, 20, 23] is that we do not need a tactic to automatically generate reified formulae.

We prove in Coq that `equivf` is reflexive, symmetric and transitive. We make the equivalence structure of `equivf` a canonical structure.

The certification of `equivf` is a consequence of the certifications of `rcf_sat` and `nquantify`. We certify the universal block quantification as follows. Evaluating the formula `nquantify a k Forall f` in an environment is not affected by the values associated by this environment to variables X_i for $a \le i$; variables X_i for $a \le i < a+k$ are bound by the block quantification and variables X_i or $a+k \le i$ are evaluated to 0 by definition of our block quantification. Thus, we consider environments `e` whose size is exactly a. We formally prove:

```
Lemma nforallP (k : nat) (e : seq R) (f : formula R) :
forall v : k.−tuple R, holds (e ++ v) f
<−> holds e (nquantify (size e) k Forall f).
```

where `++` denotes the sequences concatenation. This lemma states that the evaluation of the universal block quantification over `f` in the environment `e` is true if, and only if, the evaluation of `f` is true in any environment `e ++ v` where `v` is a sequence of length `k`.

Similarly, we formally certify the existential block quantification:

```
Lemma nexistsP (k : nat) (e : seq R) (f : formula R) :
exists v : k.−tuple R, holds (e ++ v) f
<−> holds e (nquantify (size e) k Exists f).
```

4 Formalisation of Semi-Algebraic Sets

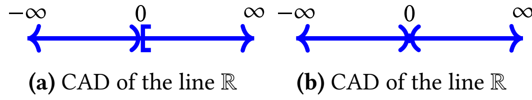
A *S.A.* set of $\mathbb{R}^n$ is the set that realizes a formula of $\mathscr{F}_n$. It is a subset of $\mathscr{F}_n$. However, numerous elements of $\mathscr{F}_n$, some of which are quantifier-free, realize the same *S.A.* set. Testing for the equality of two *S.A.* sets amounts to testing for the equivalence of two elements of $\mathscr{F}_n$ under any environment.

We firstly explain why we need the *S.A.* set concept by introducing CAD in Sect. 4.1. Then, in Sect. 4.2, we explain how we build *S.A.* sets as the mathematical quotient of $\mathscr{F}_n$ by exploiting the logical equivalence relation over $\mathscr{F}_n$.

4.1 Cylindrical Algebraic Decomposition Definition

A CAD of $\mathbb{R}^n$ is a partition $(C_j)_{1\le j\le c}$ ($c \in \mathbb{N}^*$) of $\mathbb{R}^n$ into connected *S.A.* sets C_i, called cells, satisfying the following property: for any canonical projection $\pi : \mathbb{R}^n \longrightarrow \mathbb{R}^{n-k}$ (with $k \le n$), consisting in forgetting the last k coordinates, and for any cells C_i and C_j, one have $\pi(C_i) = \pi(C_j)$ or else $\pi(C_i) \cap \pi(C_j) = \emptyset$. The images by π of the cells define a CAD of $\mathbb{R}^{n-k}$ (note that *S.A.* sets are stable by the canonical

386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440



(a) CAD of the line $\mathbb{R}$ **(b)** CAD of the line $\mathbb{R}$

Figure 1

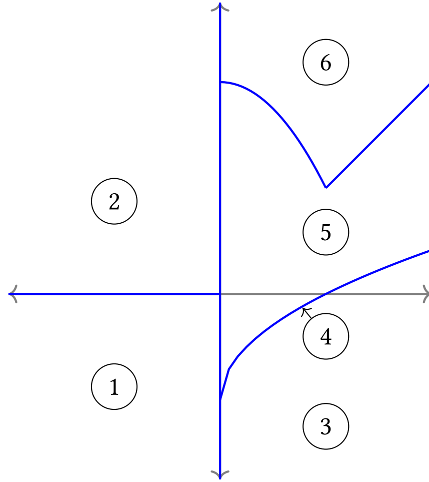


Figure 2. CAD of the plane $\mathbb{R}^2$

projections) [2, 26]. When $n = 1$, the above definition of CAD amounts to.

Another view of the CAD output is a bottom-up view, which starts by defining the CAD in dimension 1, then 2, etc. It is the original approach of Collins [5]. In dimension 1, a CAD is a finite partition of $\mathbb{R}$ into intervals (including singletons). For example, $\{]-\infty, 0[, [0, \infty[\}$ (Fig. 1a) and $\{[-\infty, 0], [0], [0, \infty]\}$ (Fig. 1b) are two valid CAD of $\mathbb{R}$. Then, one defines inductively a CAD of $\mathbb{R}^{n+1}$ by using a CAD of $\mathbb{R}^n$. A cell in the CAD of $\mathbb{R}^{n+1}$ is defined above a cell of the given CAD of $\mathbb{R}^n$ by providing one or two delimiting continuous *S.A.* functions; we define *S.A.* functions in Sect. 5. For example, the CAD (Fig. 2) defined by the cells:

- $\{(x, y) \in F^2 | x < 0, 0 < y\}$ (1)
- $\{(x, y) \in F^2 | x < 0, y \leq 0\}$ (2)
- $\{(x, y) \in F^2 | 0 \leq x, y < \sqrt{x} - 1\}$ (3)
- $\{(x, y) \in F^2 | 0 \leq x, y = \sqrt{x} - 1\}$ (4)
- $\{(x, y) \in F^2 | 0 \leq x, \sqrt{x} - 1 < y \leq 2 - x^2\}$ (5)
- $\{(x, y) \in F^2 | 0 \leq x, 2 - x^2 < y\}$ (6)

is a valid CAD of $\mathbb{R}^2$ built on top of the CAD of Fig. 1a.

Both views are based on continuous *S.A.* functions, because *S.A.* set connectedness (in the higher level view) is equivalent to *S.A.* path connectedness of *S.A.* sets, where paths are expressed with *S.A.* functions. This shows the central role played by *S.A.* functions, whose continuity is a first-order property and thus is decidable (see Sect. 7).

4.2 Definition of Semi-Algebraic Sets as a Quotient

We require that *S.A.* sets have the standard equality of the system (= in Coq). It is not always possible to build properly a quotient with the standard equality in Coq [8]. One possibility would be to compute a canonical equivalent representative of any formula, such that we compute the same representative for any two equivalent formulae.

Instead, we build the quotient out of our decidable equivalence relation, exploiting the second solution of Cohen [8] provided in the generic quotient module of Mathematical Components. *S.A.* sets is the quotient of $\mathscr{F}_n$ by the logical equivalence relation (that we defined in Sect. 3):

```
Definition SAtype := {eq_quot sub_equivf}.
```

We create a notation in Coq to denote *S.A.* sets of $\mathbb{R}^n$ by: `{SAset F^n}`.

This construction is possible because the base type $\mathscr{F}_n$ has a choice structure (`choiceType` in Coq) and the equivalence relation is decidable (that is it returns a `bool` in Coq).

The equality of two *S.A.* sets `s1` and `s2` means that their underlying formulae are logically equivalent; or, equivalentally: $\forall x \in \mathbb{R}^n\ x \in s1 \iff x \in s2$ (written s1 =i s2 in Coq). We formally prove the equivalence of *S.A.* sets viewed as formulae and viewed as sets, which states:

```
Lemma SAsetP (s1 s2 : {SAset R^n}) :
  reflect (s1 =i s2) (s1 == s2).
```

We equip *S.A.* sets with a lattice structure with bottom and top elements by implementing the `porder` interface from the developement version of Mathematical Components [6]. Specifically, we define the elements: empty, singleton, bottom and top; and the decidable operations: inclusion order, meet and join. We exploit the lemma `SAsetP` and formally prove that meet and join are associative and commutative, that the inclusion is reflexive, antisymmetric and transitive. We formally prove the distributivity of meet over join.

5 Reification Problem for the Functionality Property in the Formalisation of Semi-Algebraic Functions

In Sect. 4.1, we have used continuous *S.A.* functions to give a bottom-up definition of CAD. A *S.A.* function is a function whose graph is a *S.A.* set. More precisely, let n and $m \in \mathbb{N}$ and a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The graph of f is the set $\{(x, y) \in \mathbb{R}^n \times \mathbb{R}^m | f(x) = y\} \subseteq \mathbb{R}^{n+m}$. We define a *S.A.* function from $\mathbb{R}^n$ to $\mathbb{R}^m$ as a *S.A.* set G of $\mathbb{R}^{n+m}$ that satisfies the two following properties:

- it is total with respect to n and m (`total_SAset`).
- it is functional with respect to n and m (`funct_SAset`)

We pack n consecutive universal (resp. existential) quantifications over $\mathbb{R}$ into one universal (resp. existential) quantification over $\mathbb{R}^n$ (block quantification). The totality of G is

551 expressed through a first-order formula:
552 $\forall x \in \mathbb{R}^n \, \exists y \in \mathbb{R}^m \; (x, y) \in G$.
553 The functionality of G is then expressed through a first-order
554 formula:
555 $\forall x \in \mathbb{R}^n \, \forall y \in \mathbb{R}^m \, \forall z \in \mathbb{R}^m \; (x, y) \in G \land (x, z) \in G \implies y = z$
556 (in the code snippet 6.2 we use this property to certify the
557 Coq predicate `SAfunc`). We define *S.A.* functions in Coq by:

```
Record SAfun := MkSAfun
{
  SAgraph :> {SAset R^(n + m)};
  _ : (SAgraph \in total_SAset) && (SAgraph \in funct_SAset)

}.
```

In Coq, such a definition of `SAfun` as a subtype of `SAset` requires the decidability of the graphs's property. Then, the Hedberg theorem applies to the type representing the graphs's property, so that this type is uniquely inhabited. From this uniqueness we get that deciding the structural equality between two *S.A.* functions comes down to deciding the equality between two *S.A.* sets.

In Coq, we represent $\mathbb{R}^n$ by the vectors `'rV[R]_n` from Mathematical Components. The totality of G is expressed by the term `P` in `Prop`:

```
forall (x : 'rV[R]_n), exists (y : 'rV[R]_m), row_mx x y \in G
```

and the functionality of G is then expressed by the term `Q` in `Prop`:

```
forall (x : 'rV[R]_n), forall (y z : 'rV[R]_m),
row_mx x y \in G -> row_mx x z \in G -> y = z
```

where `row_mx` denotes the vectors concatenation.

We apply the reification technique presented in Sect. 3 to the totality property (resp. the functionality property) in order to express P (resp. Q) in a decidable way.

In the language of real closed fields, we need n variables to represent `x`, m variables to represent `y` and m variables to represent `z`. We choose to represent `x` by the consecutive variables `'X_0`, …, `'X_(n - 1)`, `y` by the consecutive variables `'X_n`, …, `'X_(n + m - 1)` and `z` by the consecutive variables `'X_(n + m)`, …, `'X_(n + 2*m - 1)` (see Fig. 3).

Since the definition of the totality (resp. functionality) property is in a prenex normal form, we achieve the reification of the totality (resp. functionality) in two steps. Firstly, we reify the quantifier-free part of the property. Finally, we apply block quantification on the resulting formula.

We start with building a tailored reified formula representing `row_mx x y \in G` (which does not use the variable `z`). The underlying formula `f` of `G` (obtained by forgetting the constraint on its free variables) already does the job, by definition (see Fig. 4a). We complete the reification of the totality of G by adding the quantifiers, in Coq:
`nquantify 0 n Forall (nquantify n m Exists f).`

605

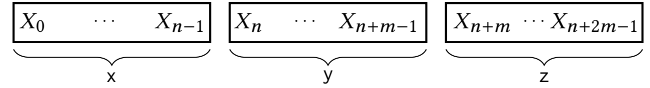
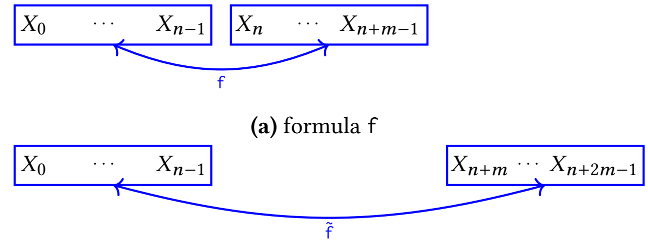


Figure 3. representation of variables `x`, `y` and `z`



(a) formula `f`

(b) formula `f̃` resulting from renaming variables in `f`

Figure 4

The reification of Q is problematic. If we keep the same representation for variables `x`, `y` and `z`, then $(x, y) \in G$ still directly reifies to (the underlying formula of) `f`. However, we cannot directly use `f` to express $(x, z) \in G$, because `z` is expected to be represented by `'X_n`, …, `'X_(n + m - 1)`, not by `'X_(n + m)`, …, `'X_(n + 2*m - 1)`. In Sect. 6, we show how to reify $(x, z) \in G$ and Q by performing substitutions in formula.

6 Use of Substitution to reify the Functionality Property

Consider the formula `f̃` obtained by renaming the variables `'X_n`, …, `'X_(n + m - 1)` in `f` (see Fig. 4a) to the respective variables `'X_(n + m)`, …, `'X_(n + 2*m - 1)` (see Fig. 4b). Using the bracket notation for simultaneous substitution, we have:
`f̃ = f['X_n / 'X_(n + m), . . . , 'X_(n + m - 1) / 'X_(n + 2*m - 1)]`.

Then, the subterm $(x, z) \in G$ of P directly reifies to the formula `f̃` itself. The subterm $(x, y) \in G \land (x, z) \in G$ of P thus reifies to `f /\ f̃`, and the proposition $\forall x \in \mathbb{R}^n \, \forall y \in \mathbb{R}^m \, \forall z \in \mathbb{R}^m \; (x, y) \in G \land (x, z) \in G$ reifies to:

```
nquantify 0 (n + 2*m) Forall (f /\ f̃).
```

For the moment, let's admit that $y = z$ is represented by a reified formula `eq_vec` (miming $\bigwedge_{i=n}^{n+m-1} X_i = X_{i+m}$). Then, the property P finally reifies to:

```
nquantify 0 (n + 2*m) Forall (f /\ f̃) ==> eq_vec.
```

We explain how we formalise the substitution in Sect. 6.1 and how to formalise `f`, `f̃` and `eq_vec` in Sect. 6.2, which completes the formalisation of the functionality property.

661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715

6.1 A new Formalisation of Substitution in Formulae

We have used simultaneous substitution of variables for terms in the formula f to define $\tilde{f}$. We call it a "block substitution", because all substitution variables (from 0 to n - 1) form a block of consecutive variables. In all our reifications, we will use block substitutions only.

The formalisation of the substitution in formulae is tricky: one should operate substitution for free occurences of variables only. Moreover, one should prevent free variables in introduced terms from falling under the scope of quantifiers, which is solved by alpha conversion [13, 14]. Simultaneity in substitution does not pose any difficulty.

Instead, we use a new, simpler, formalisation of the block substitution exploiting the formalisation of the quantifier elimination. Firstly, we define the substitution of variables for terms in a given term, in a way similar to [14]. Secondly, based on the substitution in term, we define the substitution of variables for terms in a given formula. We avoid alpha conversion by exploiting quantifier elimination. To the knowledge of the author, it is the first time that the simultaneous substitution of variables for terms in a formula is formalised in this way.

Consider the substitution t[s_0/ 'X_0, . . . , s_{p-1}/ 'X_(p - 1)] where t, s_0, …, s_{p-1} are terms and p is a natural number. Let 'X_(n - 1) be the free variable in t with highest index. When $n \leq p$ the above substitution boils down to t[s_0/ 'X_0, . . . , s_{n-1}/ 'X_(n - 1)]. When $p < n$, for convenience, variables with index greater than p are assigned to 0. That is, we formalise the following slighlty modified version of substitution:
t[s_0/ 'X_0, . . . , s_{p-1}/ 'X_(p - 1), 0 / 'X_p, . . . , 0 / 'X_(n - 1)].
Since the variable names are natural numbers, the mapping from variable names to terms can be encoded by a sequence of terms. In Coq, we define the substitution subst_term of terms for variables in a given term t with:

```
Definition subst_term (s : seq (term F)) :=
let fix sterm (t : term F) := match t with
| 'X_i => if (i < size s) then (nth 'X_0 s i)
    else 0
| t1 + t2 => (sterm t1) + (sterm t2)
| - t => - (sterm t)
| t *+ i => (sterm t) *+ i
| t1 * t2 => (sterm t1) * (sterm t2)
| t ^-1 => (sterm t) ^-1
| t ^+ i => (sterm t) ^+ i
| _ => t
end in sterm.
```

With this encoding, it is possible to keep variables unchanged by assigning their own index explicitly. For example, the substitution t[t_1/ 'X_2, t_2/ 'X_4] where variables with index greater than 5 are assigned to 0 is expressed by: subst_term [::'X_0 ; 'X_1 ; t1 ; 'X_3 ; t2 ; 'X_5]%oT t. We formally prove the correctness property for the function subst_term:

716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770

```
Lemma eval_subst e (s : seq (term F)) (t : term F) :
eval e (subst_term s t) =
eval [::]
(subst_term [seq (subst_term [seq x%oT | x <- e] u) | u <- s] t).
```

This lemma reads as follows. Evaluating a substituted term subst_term s t in an environment e boils down to evaluating the substitutors s in an environment e (which outputs [seq (subst_term [seq x%oT | x <- e] u)| u <- s]) then operate the substitution with the new substitutors, and finally evaluating the result in the empty environment.

In Coq, we define the substitution in formulae in two steps. Firstly, we define a function qf_subst_formula which performs substitution in quantifier free formulae; in this case there is no bound variables problem. In Coq, the function qf_subst_formula is defined by:

```
Fixpoint qf_subst_formula s (f : formula F) :=
let sterm := subst_term s in
match f with
| (t1 == t2) => (sterm t1) == (sterm t2)
| t1 <% t2 => (sterm t1) <% (sterm t2)
| t1 <=% t2 => (sterm t1) <=% (sterm t2)
| Unit t => Unit (sterm t)
| f1 /\ f2 => (qf_subst_formula s f1) /\ (qf_subst_formula s f2)
| f1 \/ f2 => (qf_subst_formula s f1) \/ (qf_subst_formula s f2)
| f1 ==> f2 => (qf_subst_formula s f1) ==> (qf_subst_formula s f2)
| ~ f => ~ (qf_subst_formula s f)
| ('forall 'X_i, _) | ('exists 'X_i, _) => False
| _ => f
end%oT.
```

When applied to quantification formulae, qf_subst_formula returns the arbitrary default False formula. Secondly, we define the substitution in formulae by combining qf_elim and qf_subst_formula:

```
Definition subst_formula s (f : formula F) :=
qf_subst_formula s (qf_elim f).
```

We formally prove the correctness property for the function subst_formula:

```
Lemma holds_subst e s f :
holds e (subst_formula s f)
<-> holds [::] (subst_formula
    [seq (subst_term (map Const e) t) | t <- s] f).
```

This lemma reads as follows. Evaluating a substituted term subst_term s f in an environment e boils down to evaluating the substitutors s in an environment e (which ouputs

`[seq (subst_term (map Const e)t)| t <- s]` then operate the substitution with the new substitutors, and finally evaluating the result in the empty environment.

6.2 Complete Formalisation of the Functionality Property

We are now able to detail the whole reification of the functionality property, by formalising $\tilde{f}$ with the help of `subst_formula` The formula $\tilde{f}$ rewrites as:

```
subst_formula (map (@Var _) (iota 0 n ++ iota (n + m) m)) f
```

where `iota 0 n ++ iota (n + m)m` is the concatenation of the sequences $(0, \cdots, n-1)$ and $(n+m, \cdots, n+2*m-1)$. We get the expected substitution for the following reasons. The variables `'X_0`, ..., `'X_(n - 1)` are respectively replaced by `'X_0`, ..., `'X_(n - 1)`, which has no effect. Then the variables `'X_n`, ..., `'X_(n + m - 1)` are respectively replaced by variables `'X_(n + m)`, ..., `'X_(n + 2*m - 1)`, which is what we intend to do.

Finally, we need to express the equality of y and z, that is the equality of the variable `'X_i` with `'X_(i + m)` for $i \in n, \cdots, n+m-1$, which rewrites as $\bigwedge_{i=n}^{n+m-1} X_i = X_{i+m}$, which rewrite as $\bigwedge_{i=0}^{m-1} X_{u_i} = X_{v_i}$ where $u = (n, \cdots, n+m-1)$ and $v = (n+m, \cdots, n+2*m-1)$. We view this conjonction of equalities as a binary relation between two blocks of consecutive variables, the first one ranging from `n` to `n + m - 1`, the second one ranging from `n + m` to `n + 2*m - 1`. In Coq, we express this binary relation by:

```
Definition eq_vec (v1 v2 : seq nat) : formula R :=
if size v1 == size v2
then
  (\big[And/True]_(i < size v1)
    ('X_(nth 0%N v1 i) == 'X_(nth 0%N v2 i)))%oT
else False%oT.
```

We choose that `eq_vec` returns the `formula` `False` when the input sequences have different lengths. We formally prove the correctness property for the function `eq_vec`:

```
Lemma holds_eq_vec e v1 v2 :
holds e (eq_vec v1 v2) <-> subst_env v1 e = subst_env v2 e.
```

We have now all the bricks to reify the functionality property:

```
Definition functional (f : {formula n + m}) :=
(nquantify 0 (n + 2*m) Forall
((f /\ (subst_formula (iota 0 n ++ iota (n + m) m) f))
    ==> (eq_vec (iota n m) (iota (n + m) m)))).
Definition funct_SAset : pred {SAset F ^ (m + n)} :=
    [pred s | rcf_sat [::] (functional s)].
```

We can replace `f` by the equivalent formula `(subst_formula (iota 0 n ++ iota n m) f)` in the definition of `functional`, to simplifies the correctness proof of `functional`.

We formally prove the correctness property of `SAfunc`:

```
Lemma SAfuncE (s : {SAset F ^ (n + m)}) :
reflect
(forall (x : 'rV[F]_n), forall (y1 y2 : 'rV[F]_m),
(row_mx x y1) \in s -> (row_mx x y2) \in s -> y1 = y2)
(s \in SAfunc).
```

6.3 Block Reasoning

We call block reasoning the methodology used to express the functionality property in a decidable way (this methodology is partially applied to express `equivf`). Block reasoning may apply to other first-order properties in the language of real closed fields. Block reasoning is divided into two parts. In the first part, we create a tailor-made reified formula, eventually exploiting block substitution (`subst_formula`) and block quantification (`nquant`). In the second part, we prove the correctness of the created formula, eventually exploiting environment substitution (`subst_env`) or certification of block quantification (`nforallP`) and (`nexistsP`).

We apply block reasoning to formalise the composition of two *S.A.* functions; as summarised in Sect. 7.
In Sect. 8, we explain how we use this methodology to express the continuity of two *S.A.* functions, which turns out to be more difficult than the formalisation of the composition. Reifying the continuity property leads us to extend block reasoning. We achieve the first part for the continuity property.

In all our methodology case studies, we use block substitution only to rename variables. That is, we substitute terms, which are variables only, for variables in formulae. Instead of providing a variable sequence, we simply provide an integer sequence. For convenience, we use a specialised version of block substitution, as follows:

```
Definition subst_term (s : seq nat) :=
let fix sterm (t : GRing.term F) := match t with
| 'X_i => if (i < size s)%N then 'X_(nth O s i)
    else 0
| t1 + t2 => (sterm t1) + (sterm t2)
| - t => - (sterm t)
| t *+ i => (sterm t) *+ i
| t1 * t2 => (sterm t1) * (sterm t2)
| t ^-1 => (sterm t) ^-1
| t ^+ i => (sterm t) ^+ i
| _ => t
end%T in sterm.
```

```
Fixpoint qf_subst_formula (s : seq nat) (f : formula F) :=
let sterm := subst_term s in
```

```
match f with
| (t1 == t2) => (sterm t1) == (sterm t2)
| t1 <% t2 => (sterm t1) <% (sterm t2)
| t1 <=% t2 => (sterm t1) <=% (sterm t2)
| Unit t => Unit (sterm t)
| f1 /\ f2 => (qf_subst_formula s f1) /\ (qf_subst_formula s f2)
| f1 \/ f2 => (qf_subst_formula s f1) \/ (qf_subst_formula s f2)
| f1 ==> f2 => (qf_subst_formula s f1) ==> (qf_subst_formula s f2)
| ~ f => ~ (qf_subst_formula s f)
| ('forall 'X_i, _) | ('exists 'X_i, _) => False
| _ => f
end%oT.
```

7 Application of Block Reasoning to the Composition of two Semi-Algebraic Functions

In Coq, *S.A.* functions are viewed as functions by declaring a coercion:

```
Coercion safun_to_fun : SAfun >-> Funclass.
```

Given f of type {SAfun R^m -> R^n}, g of type {SAfun R^n -> R^p} and x of type 'rV[R]_m, this allows us to write f x (of type 'rV[R]_n) and the composition g \o f. However, the latter term has type 'rV[R]_m -> 'rV[R]_p, whereas the composition of two *S.A.* functions is also a *S.A.* function. To solve this problem, we use block reasoning as in Sect. 5 and build a formula SAcomp_graph f g that represents g of as a *S.A.* set and prove its correctness:

```
Lemma SAcomp_graphP (m n p : nat)
(f : {SAfun R^m —> R^n}) (g : {SAfun R^n —> R^p})
(u : 'rV[R]_m) (v : 'rV[R]_p) :
(row_mx u v \in SAcomp_graph f g) = (g (f u) == v).
```

Based on this lemma, we prove that our formula compo f g is both functional and total, which enables us to build the *S.A.* composition sa_comp f g. Finally, we prove the correctness of the *S.A.* composition with respect to the function composition:

```
Lemma SAcompP (m n p : nat)
(f : {SAfun R^m —> R^n}) (g : {SAfun R^n —> R^p}) :
SAcomp f g =1 g \o f.
```

that is, the *S.A.* composition is the composition of functions.

8 Application of Block Reasoning to reify the Continuity of two Semi-Algebraic Functions

Block reasoning also applies to the continuity of a *S.A.* function. Since all norms are equivalent in finite dimension, we can express continuity with the supremum norm $\|.\|_\infty$:
$\forall x \in \mathbb{R}^n \forall \epsilon \in \mathbb{R} \exists \eta \in \mathbb{R} \forall y \in \mathbb{R}^n \ \|x-y\|_\infty \le \eta \implies \|f(x)-f(y)\|_\infty \le \epsilon$
where the subtraction and the sup functions are semi-algebraic. As a result, we can prove that the continuity of a *S.A.* function is a decidable property.

We show how to use block reasoning to produce a reified formulae for the continuity of two *S.A.* functions. The certification of the resulting formula is still a work in progress.

Since the definition of the continuity is in a prenex normal form, we achieve the reification of the continuity in two steps. Firstly, we reify the quantifier-free part of the continuity property. Finally, we apply block quantification on the resulting formulae.

Reifying the quantifier-free part of the continuity property boils down to reifying $\kappa = \|x-y\|_\infty \le \eta \implies \|u-v\|_\infty \le \epsilon$ and adding the relations $f(x) = u$ and $f(y) = v$ — once variables 'X_i are chosen to represent x and u (resp. y and v), we already know how to represent $f(x) = u$ (resp. $f(y) = v$), by using substitution.

The language of reified formulae already has a constant (==> in Coq) to express the logical implication. Thus, reifying κ boils down to reifying $\|x-y\|_\infty \le \eta$ and $\|u-v\|_\infty \le \epsilon$. Indeed, the reification of $\|u-v\|_\infty \le \epsilon$ is obtained by reifying $\|x-y\|_\infty \le \eta$ for m instead of n and renaming variables. Moreover, the language of reified formulae already has a constant (<=% in Coq) to express the inequality in $\mathbb{R}$. Thus, reifying κ boils down to reifying $\|x-y\|_\infty$, which we decompose into three steps. We separately reify subtraction in $\mathbb{R}^n$, coordinate-wise absolute value in $\mathbb{R}^n$ and the coordinate maximum function from $\mathbb{R}^n$ to $\mathbb{R}$. We combinate these three constructions to reify $\|x-y\|_\infty$.

Given variables blocks representing x and y, we firstly create a formula (8) representing the subtraction $x-y$. This formula has a third variables block to represent the output $x-y$ and a subformula to express the ternary relation between x, y and $x-y$. Then, we create a formula (8) representing coordinate-wise absolute value. This formula has a block to represent z, a block to represent the vector $(|z_i|)_i, (0 \le i < n)$, and a subformula to express the binary relation between z and $(|z_i|)_i$. We apply it to represent the vector $|x_i - y_i|$ $(0 \le i < n)$. Finally, we create a formula (8) representing the coordinates maximum of a vector. This formula has a block to represent z, a single variable to represent $\max_{0\le i<n} z_i$ and a subformula to represent the relation between z and $\max_{0\le i<n} z_i$. We apply it to represent $\|x-y\|_\infty$.

Block Reasoning Extension In this paragraph, we extend block reasoning with block subtraction (8), block absolute value (8) and block maximum (8).

These constructions are parameterised by the desired sequences of variables

```
Definition sub_vec (v1 v2 v3 : seq nat) : formula F :=
  if ((size v1 == size v2) && (size v1 == size v3))
    then \big[And/True]_(i < size v1) ( 'X_(nth 0 v1 i) ==
           'X_(nth 0 v2 i) + 'X_(nth 0 v3 i))%oT
    else False.

Definition abs_vec (v1 v2 : seq nat) : formula F :=
  if size v1 == size v2
    then (\big[And/True]_(i < size v1) (abs (nth 0 v1 i)) (
           nth 0 v2 i))%oT
    else False.

Definition max_vec (v : seq nat) (n : nat) : formula F :=
((\big[Or/False]_(i < size v) ('X_n == 'X_(nth 0 v i))) /\
(\big[And/True]_(i < size v) ('X_(nth 0 v i) <=% 'X_n)))%oT.
```

We also prove the following correctness properties for `sub_vec` and `abs_vec`, in Coq:

```
Lemma sub_vecP (e : seq F) (v1 v2 v3 : seq nat) :
rcf_sat e (sub_vec v1 v2 v3) = ((size v1 == size v2) && (size v1
        == size v3)) &&
[forall i,
(e_(nth 0 v3 (i : 'I_(size v1))) == e_(nth 0 v1 i) - e_(nth 0 v2 i))].


Lemma abs_vecP (e : seq F) (v1 v2 : seq nat) :
rcf_sat e (abs_vec v1 v2) = (size v1 == size v2)
              && [forall i, e_(nth 0 v2 (i : 'I_(size v1))) == |e
                 _(nth 0 v1 i)|].
```

We now come back to the reification of κ. ⧫

To reify κ, each *S.A.* function application occurring in κ (subtraction, absolute value, maximum and `f`) requires variables `'X_i` to represent both its inputs and its ouput. Let denote the output dimension of f by p, in other words f has type $\mathbb{R}^n \to \mathbb{R}^p$.

We need n variables to represent x, n variables to represent y, n variables to represent $x-y$, n variables to represent the row $|x_i - y_i|$ $(0 \leq i < n)$, one variable to represent $\|x-y\|_\infty = \max_{0 \leq i < n} |x_i - y_i|$ and one variable to represent η. Similarly, we need m variables to represent u, m variables to represent v, m variables to represent $u - v$, m variables to represent the row $|u_i - v_i|$ $(0 \leq i < n)$, one variable to represent $\|u-v\|_\infty = \max_{0 \leq i < n} |u_i - v_i|$ and one variable to represent ϵ.
We choose to represent x by the consecutive variables `'X_0 ..., 'X_(n - 1)`, y by the consecutive variables `'X_n, ..., 'X_(2*n - 1)`, $x-y$ by the consecutive variables `'X_(2*n), ..., 'X_(3*n - 1)`, $|x_i - y_i|$ $(0 \leq i < n)$ by the consecutive variables `'X_(3*n), ..., 'X_(4*n - 1)`, $\|x-y\|_\infty$ by the variable `'X_(4*n)` and η by the variable `'X_(4*n + 1)` (see Fig. 5b).

We use a similar variable representation for the subformula $\|u-v\|_\infty \leq \epsilon$, starting at index `4*n + 2`. We choose to represent u by the consecutive variables `'X_0 ..., 'X_(n - 1)`, v by the consecutive variables `'X_n, ..., 'X_(2*n - 1)`, $u - v$ by the consecutive variables `'X_(2*n), ..., 'X_(3*n - 1)`, $|u_i - v_i|$ $(0 \leq i < n)$ by the consecutive variables `'X_(3*n), ..., 'X_(4*n - 1)`, $\|u-v\|_\infty$ by the variable `'X_(4*n)` and ϵ by `'X_(4*n + 1)` (see Fig. 5c).

Indeed, the reification of $\|u-v\|_\infty \leq \epsilon$ is obtained by reifying $\|x-y\|_\infty \leq \eta$ for m instead of n and by renaming variables by shifting indices by $4n+2$ positions.

By exploiting block reasoning on our variable representation, $\|x-y\|_\infty \leq \eta$ reifies to the following Coq formula applied to n:

```
Definition bloc (i : nat) : formula F :=
  (sub_vec (iota 0 i) (iota i i) (iota (2*i) i))
  /\ (abs_vec (iota (2*i) i) (iota (3*i) i))
  /\ (max_vec (iota (3*i) i) (4*i))
  /\ ('X_(4*i) <=% 'X_((4*i).+1)).
```

To reify $\|u-v\|_\infty \leq \epsilon$, we thus rename variables `'X_0 ..., 'X_(4*m + 1)` to `'X_(4*n + 2) ..., 'X_(4*n + 4*m + 3)` in `bloc m`, with block substitution: $\|u-v\|_\infty \leq \epsilon$ reifies to:

```
subst_formula (iota (4*n + 2) (4*m + 1)) (bloc m)
```

thus, κ reifies to:

```
(bloc n) ==> (subst_formula (iota (4*n + 2) (4*m + 1)) (bloc m))
```

which we denote by `beta`. We now have to restrict κ to values x, y, u and v such that $f(x) = u$ and $f(y) = v$. We already know how to reify the latter equations by applying block substitutions on f; we reify the quantifier-free part of the continuity property in Coq with:

```
beta /\ (subst_formula ((iota 0 n) ++ (iota (4*n + 2) m)) f)
     /\ (subst_formula ((iota n n) ++ (iota (4*n + m + 2) m)) f).
```

which we denote by `gamma`.

We now achieve the full reification of the continuity property by adding all the quantifications with:

```
Definition is_continuous_form (f : {formula_(n + m) F}) :=
nquantify 0 n Forall
 ('forall 'X_(4*n + 4*m + 3),
  ('exists 'X_(4*n + 1), (nquantify n (2*n) Forall
   (nquantify (2*n) (2*n + 1) Forall
    (nquantify (4*n + 2) (4*m + 1) Forall gamma))))).
```

9 Related Work

The main related work is the book *Algorithms In Real Algebraic Geometry* [2], which describes various algorithms including "CAD algorithms". While this book contains "paper" algorithms and proofs, it comes with implementations

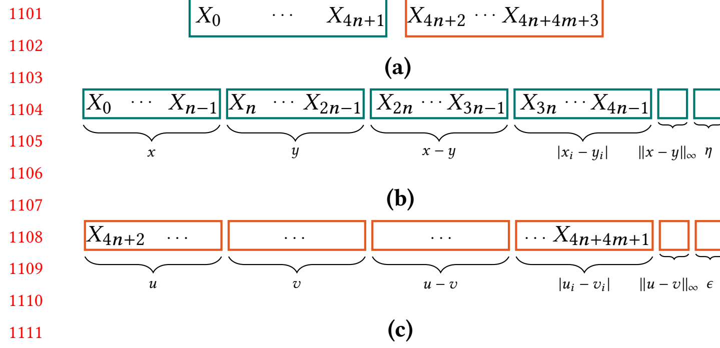


Figure 5. representation of the input variables, intermediate variables and output variables of the continuity property

of some algorithms in Maxima [4]. We use this book as our main reference.

Formal proofs of real algebraic geometry theory and algorithms in a proof assistant started, up to our knowledge, with Mahboubi's formalisation of Collins' "CAD algorithms" [19]. She certified the key component which computes Sturm sequences efficiently using subresultants. Then, Cohen and Mahboubi [9] formalised a quantifier elimination algorithm, following one of the many algorithms in *Algorithms In Real Algebraic Geometry* [2] with an approach similar to Tarski's initial proof [24]. The latter formalisation relies on algebraic arguments, which makes it a solid ground for our current work.

The first closest formal proof to our work was John Harrison's HOL Light [15] quantifier elimination procedure, but it is viewed as a decision procedure and does not tackle real algebraic geometry theory. There have been several other formalisations of Sturm theorem, in HOL Light [12] and then in PVS [22] and concurrently in Isabelle/HOL [10, 25]. Also there is a proof of Rouché Theorem [17] to tackle another aspect of "CAD algorithms".

The work we present here is different from other formalisations because it does not focus on the algorithmic ingredients present in "CAD algorithms" but rather on the theory that is necessary to describe its result precisely.

Conclusion and Future Work

We have shown how we handle *S.A.* sets and *S.A.* functions concepts in Coq for the needs of the CAD formalisation. The methods we use to formalise and certify functionality and totality of *S.A.* functions also apply to the continuity of a *S.A.* function. This paves the way for the formalisation of continuous *S.A.* functions and thus the CAD output.

In a straight continuation of our construction of the composition of two *S.A.* functions, we should be able to provide other *S.A.* functions, such as: polynomials, linear functions and the nth virtual roots applications (as defined by Gonzalez-Vega, Lombardi, & Mahé [11]).

Acknowledgments

Pretty Coq code listing was done thanks to Assia Mahboubi's file [18]. We thank our colleagues for their input on this work: Assia Mahboubi, Cyril Cohen, my co-supervisor, Damien Rouhling, Enrico Tassi, José Grimm, Laurent Théry, Matej Košík, Yves Bertot, my supervisor, and the MAP working group about real geometry, including Marie-Françoise Roy and Henri Lombardi.

References

[1] J. R. Johnson B. F. Caviness (Ed.). 1998. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* https://doi.org/10.1007/978-3-7091-9459-1

[2] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. 2006. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[3] Christopher W. Brown. 2003. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin* 37 (November 2003), 97–108. Issue 4.

[4] Fabrizio Caruso. 2006. *The SARAG Library: Some Algorithms in Real Algebraic Geometry.* Springer Berlin Heidelberg, Berlin, Heidelberg, 122–131. https://doi.org/10.1007/11832225_11

[5] B.F. Caviness and J.R. Johnson. 2012. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer Vienna. https://books.google.fr/books?id=vu-pCAAAQBAJ

[6] Cyril Cohen. [n. d.]. Finmap Library. https://github.com/math-comp/finmap. ([n. d.]). Accessed: 07/04/2017.

[7] Cyril Cohen. 2012. Construction of real algebraic numbers in Coq. (2012).

[8] Cyril Cohen. 2013. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings.* 213–228. https://doi.org/10.1007/978-3-642-39634-2_17

[9] Cyril Cohen and Assia Mahboubi. 2010. A Formal Quantifier Elimination for Algebraically Closed Fields. In *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings.* 189–203. https://doi.org/10.1007/978-3-642-14128-7_17

[10] Manuel Eberl. 2015. A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP '15).* ACM, New York, NY, USA, 75–83. https://doi.org/10.1145/2676724.2693166

[11] Laureano Gonzalez-Vega, Henri Lombardi, and Louis Mahé. 1998. Virtual roots of real polynomials. *Journal of Pure and Applied Algebra* 124, 1 (1998), 147 – 166. https://doi.org/10.1016/S0022-4049(96)00102-8

[12] John Harrison. 1997. Verifying the accuracy of polynomial approximations in HOL. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97* (19–22 August 1997) *(Lecture Notes in Computer Science)*, Elsa L. Gunter and Amy Felty (Eds.), Vol. 1275. Springer-Verlag, Murray Hill, NJ, 137–152.

[13] John Harrison. 1998. Formalizing Basic First Order Model Theory. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98* (September/October 1998) *(Lecture Notes in Computer Science)*, Jim Grundy and Malcolm Newey (Eds.), Vol. 1497. Springer-Verlag, Canberra, Australia, 153–170.

[14] John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press.

[15] John Harrison. 2011. *Theorem Proving with the Real Numbers* (1st ed.). Springer Publishing Company, Incorporated.

[16] Stéphane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement*

d'une tactique reflexive pour la demonstration automatique en coq). Ph.D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00713668

[17] Wenda Li and Lawrence C. Paulson. 2016. *A Formal Proof of Cauchy's Residue Theorem.* Springer International Publishing, Cham, 235–251. https://doi.org/10.1007/978-3-319-43144-4_15

[18] Assia Mahboubi. [n. d.]. lstcoq.sty file which defines a Coq - SSReflect style for listings in Latex. https://hal.inria.fr/file/index/docid/611757/filename/lstcoq.sty. ([n. d.]). Accessed: 08/02/2017.

[19] Assia Mahboubi. 2006. *Contributions à la certification des calculs dans R : théorie, preuves, programmation. (Contributions to the certification of computations in R : theory, proofs, implementation).* Ph.D. Dissertation. University of Nice Sophia Antipolis, France. https://tel.archives-ouvertes.fr/tel-00117409

[20] Assia Mahboubi. 2007. Implementing the Cylindrical Algebraic Decomposition Within the Coq System. *Mathematical. Structures in Comp. Sci.* 17, 1 (Feb. 2007), 99–127. https://doi.org/10.1017/S096012950600586X

[21] Assia Mahboubi and Enrico Tassi. [n. d.]. Mathematical Components. https://math-comp.github.io/mcb/. ([n. d.]). Accessed: 06/04/2017.

[22] Anthony Narkawicz, César Muñoz, and Aaron Dutle. 2015. Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems. *Journal of Automated Reasoning* 54, 4 (2015), 285–326. https://doi.org/10.1007/s10817-015-9320-x

[23] Christine Paulin-Mohring. 2011. Introduction to the Coq Proof-Assistant for Practical Software Verification. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures.* 45–95. https://doi.org/10.1007/978-3-642-35746-6_3

[24] Alfred Tarski. 1951. A decision method for elementary algebra and geometry. *Bull. Amer. Math. Soc.* 59 (1951).

[25] Lawrence C. Paulson Wenda Li, Grant Olney Passmore. 2015. A Complete Decision Procedure for Univariate Polynomial Problems in Isabelle/HOL. (2015).

[26] Wikipedia. [n. d.]. CAD. https://en.wikipedia.org/wiki/Cylindrical_algebraic_decomposition. ([n. d.]). Accessed: 11/04/2017.