Generating certified code from formal proofs: a case study in homological algebra

Jesús Aransay, Clemens Ballarin, Julio Rubio

DOI 10.1007/s00165-009-0120-0
BCS © 2009
Formal Aspects of Computing (2010) 22: 193–213

Formal Aspects of Computing

Generating certified code from formal proofs: a case study in homological algebra

Jesús Aransay[1], Clemens Ballarin[2], Julio Rubio[1]

[1] Departamento de Matemáticas y Computación, Universidad de La Rioja, c. Luis de Ulloa s/n, 26004 La Rioja, Spain.
E-mail: jesus-maria.aransay@unirioja.es, julio.rubio@unirioja.es
[2] Institut für Informatik, Technische Universität München, Munich, Germany.
URL: http://www4.in.tum.de/~ballarin

Abstract. We apply current theorem proving technology to certified code in the domain of abstract algebra. More concretely, based on a formal proof of the *Basic Perturbation Lemma* (a central result in homological algebra) in the prover Isabelle/HOL, we apply various code generation techniques, which lead to certified implementations of the associated algorithm in ML. In the formal proof, algebraic structures occurring in the Basic Perturbation Lemma are represented in a way, which is not directly amenable to code generation with the available tools. Interestingly, this representation is required in the proof, while for the algorithm simpler data structures are sufficient. Our approach is to establish a link between the non-executable setting of the proof and the executable representation in the algorithm, which is to be generated. This correspondence is established within the logical framework of Isabelle/HOL—that is, it is formally proved correct. The generated code is applied to and illustrated with a number of examples.

Keywords: Formalized mathematics, Software certification, Code generation, Homological algebra, Isabelle/HOL

1. Introduction

In our earlier paper [ABR08] we presented a complete formalized proof of a result in homological algebra known as *Basic Perturbation Lemma* (or BPL, for short). This formal proof was a first milestone in our project to obtain certified versions of algorithms applied in a symbolic computation system named Kenzo. Kenzo [DSS99] is a symbolic computation system specialized in the field of algebraic topology; the system contains features enabling computations with topological spaces of infinite nature, and it has obtained remarkable results which were previously unknown (mainly in the computation of homology groups).

In this context, a generic project to apply formal methods to the system was undertaken some years ago. Significant results about the application of algebraic specification techniques to the algebraic structures implemented in the system have been obtained [LPR03, DLR07]. A different approach was started by us. We considered some crucial algorithms of the system and tried to apply software certification methods to them. In this task, we first considered the algorithm associated to the BPL. As previously mentioned, we obtained a formalized proof of this result in the implementation of higher-order logic available in the Isabelle proof assistant (referred to as

Correspondence and offprint requests to: J. Aransay, E-mail: jesus-maria.aransay@unirioja.es
This work has been partially supported by Ministerio de Educación y Ciencia, project MTM2006-06513.

Isabelle/HOL [NPW02]), and then we considered the possibilities of Isabelle/HOL for the generation of certified code from the previous formalization. The achievements of this research are presented in this paper.

Some considerations are worth noting before getting into details:

- Our research heavily depends on the theorem proving system we have used. Since our formalized proofs are implemented in Isabelle/HOL, the results about code generation rely on the capacities of this system for generating programs. Two different *code generation* tools are available. Our first experiments were developed with Berghofer's code generation facility [Ber03b], although we finally found the tool introduced by Haftmann and Nipkow [Haf07a, HN07] more suitable. In one of our examples, we will have to resort to HCL [Obu08] (the HOL Computing Library), an extension of Isabelle designed for obtaining trusted computations (but not code generation). We will describe some of the features of these tools relevant to our work, even when our goal is far from doing a formal comparison of them.
- Different techniques can be applied to generate certified code, depending, for instance, on the nature of the results studied. We decided to employ *code generation from formal specifications*. In this technique, the formal specifications that are implemented in a proof assistant to produce formal proofs are also used as inputs for the code generation tools, obtaining, if possible, programs satisfying the given specifications. Note that in Isabelle/HOL specifications are not necessarily executable. Another approach consists in providing specifications and proofs in a constructive type theory, as considered by Coquand and Spiwack [CS07].
- Our main concern is to propose solutions to some concrete problems in computational algebra, as will be seen in the rest of the paper. Our intention is neither to define new generic tools for software certification, nor to propose code generation recipes. Nevertheless, our case study together with the solutions proposed might be helpful to solve other problems in software certification, suggest some ideas for code generation in a mathematical setting, and point out some limitations of the code generation tools.

The methodology of our approach consists in working with two different representations of a same concept (examples of *concepts* in this paper are *group* or *matrix*). One of such representations is specially designed to prove properties. The second representation is specially suitable to generate code from it. Then, a *domain transformation* is constructed between both representations, in such a way that proofs developed in the first setting can be applied to the second one, hence ensuring the correctness of the final programs generated from the initial specifications. For instance, algebraic structures will be represented as records with an explicit carrier set for our proving purposes, but by means of type classes for code generation experiments.

Another relevant aspect of our work is that not only algorithms are proven correct: The validity of the inputs with respect to their specifications is also verified. In our examples, for instance, termination of computations cannot be ensured without validating the input of the algorithms, and some of the algebraic structures over which computations are to be carried out and their operations are not trivial. Thus, the focus of our research includes both *certified programs* and *certified computations* (formally proving the correctness of inputs and processes, and therefore ensuring the correctness of the obtained outputs).

The following example illustrates the field of application of domain transformations. In one of the instances of the BPL being presented in this paper, algebraic structures are represented by means of integer matrices (this is a natural representation of bicomplexes). Here, we again keep two different representations of matrices, one as finite sets of non-zero elements, and another one as an inductive datatype based on lists. The first one satisfies the algebraic specification (that we introduce later) of differential groups with a nilpotent homotopy operator. The second one serves the purpose of generating code from the matrix representation and operations. Then, we link both representations by means of some lemmas proving that some operations over the second representation produce same results as the required operations over the first representation. These lemmas are then used as rewrite rules that enable computations over the first representation, and also ensure the correctness of such computations. We then obtain computations that are *certified*, since every step of our process has been formally proved. This example illustrates that our approach can be applied to more general contexts than homological algebra.

The paper will be divided as follows. In Sect. 2, we introduce the mathematical definitions and results which will be required in the rest of the paper. An intuitive idea of the algorithmic nature of the BPL is also included. In Sect. 3, we present the main tools involved in the specification and proof of the BPL in Isabelle/HOL that will be relevant to our case study. In particular, two possibilities to represent algebraic structures are presented: One based on *extensible records* with explicit carrier sets (useful for implementing proofs) and another one based on *type classes* (used for code generation tasks). In this section a brief presentation of the generation tools by Berghofer [Ber03b] and Haftmann and Nipkow [Haf07a, HN07] is included as well. In Sect. 4, we present different

ways to understand the process of code generation and execution, ranging from *certified programs* to *certified computations*. In Sect. 5, we propose new specifications for the non-executable parts of our formal proof of the BPL. Then we introduce the domain transformation mapping from type classes to extensible records, which enables us to apply the previous formal proof of the BPL to the new proposed specifications. In Sect. 6, code generation is applied to the new specifications, as we will show, including some excerpts of the generated programs linked to the BPL. In Sect. 7, we present two different examples of application of the obtained code over concrete instances of the BPL. These instances have to be first proved, in Isabelle/HOL, to satisfy the algebraic specification of the algebraic structures in the BPL. The first example is simple enough to be covered with the standard Isabelle machinery. The second case, based on bicomplexes, is more difficult and involves work on matrix representations (in the line evoked previously). A conclusions and further work section and the bibliography end the paper.

2. Mathematical definitions and results

In this section, we give an overview of the BPL. We focus on a version that makes the algorithmic nature of the BPL explicit. Definitions have been taken from Rubio and Sergeraert's lecture notes on constructive algebraic topology [RS97]. In our setting, the most important concept is the one of differential group.

Definition 2.1 A *differential group* is a pair (C, d_C) where C is an abelian group and d_C is an endomorphism of C such that $d_C d_C = 0_{\text{End } C}$; d_C is called the *differential map* or *boundary operator* of C.

The boundary condition $d_C d_C = 0_{\text{End } C}$ allows us to introduce the *homology group* of a differential group.

Definition 2.2 Given a differential group (C, d_C), its *homology group*, denoted $\mathrm{H}(C, d_C)$, is defined as the quotient group $\ker d_C / \operatorname{im} d_C$.

Two differential groups can be considered homologically equivalent whenever their homology groups are isomorphic. Thus, the computation of homology groups (of differential groups canonically associated to topological spaces) is one of the main problems to be solved in algebraic topology. In our computational setting, the question is refocused on *mechanically* computing homology groups. A homology group is, by definition, an abelian group. Thus, if it is finitely generated, it can be identified by a finite series of natural numbers (namely, its rank and its torsion coefficients), thanks to the classification theorem for abelian finitely generated groups.

Now, if a homology group is known to be of *finite type* (i.e., it is finitely generated), it is natural to wonder about the *computability* of such a group (i.e., about the existence of an algorithm computing its rank and its torsion coefficients). It turns out that, in the case that the differential group (C, d_C) is free of finite type (i.e., the group C is a free abelian group generated by an explicit and finite set of generators), there is an elementary algorithm (based on diagonalizing a matrix representation of the differential map d_C with respect to the given basis) which computes the homology group $\mathrm{H}(C, d_C)$.

There are also many differential groups (coming from important constructions in algebraic topology) which are not of finite type (we will refer to them as of *infinite nature*), but whose homology groups *are* of finite type. Then, it raises the question whether that homology groups can be (theoretically or practically) computed.

Sergeraert [RS97] presented one idea to solve this problem. He suggested to look for a link from a *big* differential group (of infinite nature) (D, d_D) to a *small* one (of finite type) (C, d_C), in such a way that the link defines a canonical isomorphism between their homology groups $\mathrm{H}(D, d_D)$ and $\mathrm{H}(C, d_C)$. This link, that is usually named *reduction* (see Definition 2.4 below for details), provides us with an algorithm to compute $\mathrm{H}(D, d_D)$ through the elementary computation of $\mathrm{H}(C, d_C)$. Prior to the definition of reduction, we introduce the notion of *morphism* between differential groups.

Definition 2.3 Given (A, d_A) and (B, d_B) two differential groups, a *differential group homomorphism* $f\colon (A, d_A) \to (B, d_B)$ is a group homomorphism $f\colon A \to B$ which commutes with the differentials: $f d_A = d_B f$.

The definition of *reduction* can be now introduced.

Definition 2.4 Given two differential groups (D, d_D) and (C, d_C), a *reduction* between them is a triple of homomorphisms $(f, g, h)\colon (D, d_D) \Rightarrow (C, d_C)$ satisfying:

1. The components f and g are differential group homomorphisms, $f\colon (D, d_D) \to (C, d_C)$ and $g\colon (C, d_C) \to (D, d_D)$.
2. The component h is a group endomorphism on D, called *homotopy operator*.

3. The following relations hold:
   (a) $fg = \mathrm{id}_C$;
   (b) $gf + d_D h + h d_D = \mathrm{id}_D$;
   (c) $fh = 0_{\mathrm{Hom}\,D\,C}$;
   (d) $hg = 0_{\mathrm{Hom}\,C\,D}$;
   (e) $hh = 0_{\mathrm{End}\,D}$.

Note that morphisms are added *pointwise*: $(f + f')(x)$ is, by definition, $f(x) + f'(x)$.

Now, in order to make Sergeraert's method applicable to the computation of homology groups, it turns out to be natural to work with reductions. Algorithms have been devised that take as input reductions and produce also reductions as output. The BPL is one of such algorithms (and the most frequently used in the Kenzo system [DSS99]). Roughly speaking, the idea underlying the BPL is the following. Let $(D', d_{D'})$ be a differential group whose homology group $\mathrm{H}(D', d_{D'})$ has to be computed. Let us suppose that D' is of infinite nature, and also that another differential group (D', d_D) (formed by the same group D' but a different boundary operator d_D) and a reduction from it to a finitely generated differential group (C, d_C) are known. Then $\mathrm{H}(D', d_D)$ can be directly computed, but not $\mathrm{H}(D', d_{D'})$. Let $\delta_D = d_{D'} - d_D$. This group endomorphism can be viewed as a *perturbation* of (D', d_D): adding (perturbing) d_D by δ_D, we obtain $(D', d_{D'})$. The BPL (stated in Theorem 2.7) explains how the reduction from (D', d_D) to (C, d_C) can be *perturbed* by δ_D, producing a new reduction from $(D', d_{D'})$ to $(C, d_{C'})$ (with the same group C, but a different boundary operator from the one in (C, d_C)). Since $(C, d_{C'})$ preserves the property of being of finite type, this gives an algorithm to compute the initial homology group $\mathrm{H}(D', d_{D'})$.

Going back to the technique, it is not reasonable to expect that *any* perturbation δ_D can be used to apply the BPL algorithm. A large class of suitable perturbations is defined as follows.

Definition 2.5 Let (D, d_D) be a differential group. A *perturbation* of the differential d_D is a group endomorphism $\delta_D : D \to D$ such that $d_D + \delta_D$ is a differential for the group D. A perturbation δ_D of d_D satisfies the *local nilpotency condition* with respect to a reduction $(f, g, h) : (D, d_D) \Rightarrow (C, d_C)$ whenever the composition $\delta_D h$ is pointwise nilpotent, that is, given x an element of D there exists a natural number n such that $(\delta_D h)^n(x) = 0$, where n depends on each x in D.

The following proposition is needed in order to make the BPL statement meaningful.

Proposition 2.6 *Let $(f, g, h) : (D, d_D) \Rightarrow (C, d_C)$ be a reduction between two differential groups and δ_D a perturbation of d_D satisfying the local nilpotency condition with respect to the reduction. Then both $\phi = \sum_{i=0}^{\infty}(-1)^i(\delta_D h)^i$ and $\psi = \sum_{i=0}^{\infty}(-1)^i(h\delta_D)^i$ define endomorphisms of the abelian group D.*

The proof of the previous proposition relies on the finiteness (due to the local nilpotency) of the formal series ϕ and ψ over any given element of their domain.

The following statement corresponds to the Basic Perturbation Lemma as presented in [RS97].

Theorem 2.7 Basic Perturbation Lemma. *Let $(f, g, h) : (D, d_D) \Rightarrow (C, d_C)$ be a reduction between two differential groups and $\delta_D : D \to D$ a perturbation of the differential d_D satisfying the local nilpotency condition with respect to the reduction (f, g, h). Then, a new reduction $(f', g', h') : (D', d_{D'}) \Rightarrow (C', d_{C'})$ can be obtained, where the underlying abelian groups D and D' (resp. C and C') are the same, but the differentials are perturbed: $d_{D'} = d_D + \delta_D$, $d_{C'} = d_C + \delta_C$, where $\delta_C = f\delta_D\psi g$; $f' = f\phi$; $g' = \psi g$; $h' = h\phi$, being $\phi = \sum_{i=0}^{\infty}(-1)^i(\delta_D h)^i$, and $\psi = \sum_{i=0}^{\infty}(-1)^i(h\delta_D)^i$.*

From the statement, it can be observed that the BPL produces the explicit formulae of a reduction between two differential groups. It can be also noted that the main objects to be computed in the algorithm are the series ϕ and ψ, which enable the computation of the output reduction and differentials. Actually, if the input reduction is known, the formulae providing the output reduction exclusively rely on the input reduction (f, g, h) and in the series ϕ and ψ. Therefore, the generated code from the BPL will be related to the computation of such series. The termination of computations will become then a crucial part in our examples, since it is the only way to certify, not only the generated version of the algorithm, but also the output of such an algorithm.

Different proofs of the BPL can be found in the literature [Gug72, BL91, RS97]. An extremely detailed proof is given by the first author [Ara06, Chap. 2], where a proof as close as possible to the one that was later implemented in Isabelle [ABR08] is described.

3. Basic Isabelle infrastructure

In this section, we introduce the two main concerns to be dealt with in Isabelle/HOL and which originated from the previous mathematical results. They are related to the representation of mathematical structures (Sect. 3.1) and to the implementation of series and the local nilpotency condition (Sect. 3.2). There are two reasons for presenting these topics. First, they are the ones that will have to be modified from our previous formal proof of the BPL to be able to apply code generation. Second, the series ϕ and ψ are the objects which enable the computation of the output of the algorithm associated to the BPL. Finally, in Sect. 3.3, we will briefly comment on two different Isabelle/HOL code generation tools.

3.1. Algebraic structures

In Kenzo, algebraic structures are represented as records with fields encoding their different operations in a functional way. As an example, for a differential group we would find as fields the addition, the inverse operation, the neutral element and the boundary map. Being based on the dynamic type system of Common Lisp, Kenzo has a great flexibility to encode groups in such a way, without restrictions due to cardinality (finite and infinite groups are instances of the same record structure). The only constraint is related to the *computability* of the different components of the group (since they must be programmed as Common Lisp λ–terms). Thus, it was natural for us to choose an Isabelle/HOL representation based on records to implement the formal proof of the BPL [ABR08]. The lack of dependent types in the HOL type system was overcome working with *carriers* defined through the Isabelle/HOL `set` constructor. In this way, the expressiveness of Kenzo to represent algebraic structures is recovered in Isabelle (actually, even non-computable groups could be represented).

The idea of representing algebraic structures by means of records has been widely used. For instance, Gonthier et al. [GMR07] use a similar approach to introduce a representation of the infrastructure to formally prove the Feit-Thompson theorem. This work includes a representation of finite groups, quotient structures, group morphisms and the like in the proving assistant Coq. Although our representation in Isabelle/HOL is not restricted to finite groups, some similarities can be found. First, the representation of explicit domains of algebraic structures in terms of sets, which enables the definitions of groups in terms of subsets of groups, such as the kernel, the image set, or quotient groups. Second, the representation of morphisms between algebraic structures by means of *completions*, i.e., functions that map the elements outside their domain of definition to a distinguished element, permitting to prove extensional equality of functions [ABR04].

The expressiveness of HOL allows different representations of algebraic structures. Depending on the goals pursued, different choices can be made. Here we present two of them together with their main properties. The first one uses *extensible records* for the type definitions plus *locales* for the logical specification. The second one is based on *type classes* for type definitions, and also *locales* are (internally) used for the logical specifications.

A basic example of a record type representing an algebraic structure is given by the following *semigroup* type definition:

```
record 'a semigroup =
  carrier :: 'a set
  mult :: ['a,'a ] ⇒'a ("⊗ı")
```

In the previous definition it can be observed that the domain (or carrier) of the semigroup is represented by means of an *explicit* field (`carrier`). This representation has two relevant properties. First, it allows defining endomorphisms and homomorphisms in such a way that they both can be endowed with algebraic structures that can be treated uniformly in the setting (i.e., represented by means of record types). Second, this representation eases the definition of subsets (mainly kernels and image sets of homomorphisms) of the carriers.

Another relevant feature of this type definition is that, internally, record types are encoded as *extensible records* [NW98], which can be used to define new records by adding new fields. This feature enables us to create an algebraic hierarchy in a straightforward way and to make use of *parametric polymorphism* in order to give a generic treatment of carrier sets of different types. Parametric polymorphism is also relevant for defining homomorphisms and endomorphisms in a hierarchy of algebraic structures since it allows the same function definitions to be used over different algebraic structures.

The previous `semigroup` type definition is to be coupled with a logical specification which can be given by means of a *locale* definition. Locales [Bal04] are a module system for Isabelle which allows building, managing and merging different proof and specification contexts.

```
locale semigroup = fixes G (structure)
assumes m_closed[intro,simp]: "[[x ∈carrier G; y ∈carrier G]] ==> x ⊗ y ∈ carrier G"
and m_assoc: "[[x ∈carrier G;y ∈carrier G;z ∈carrier G]] ==> (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
```

In the previous locale a concrete name (`G`) has been given for referring to the semigroup structure in the created context. Having defined explicitly the carrier of the semigroup, it has to be stated that the multiplication is closed with respect to such carrier (`m_closed`). Finally, the associativity of multiplication (`m_assoc`) is also imposed.

The adequacy of the previous representation for proving results in abstract algebra has been shown with some major examples, such as the formal proof of the First Sylow's Theorem presented in [KP99]. We made use of this representation also in our formal proof of the BPL after having tried various different approaches (see [ABR04] for a detailed comparison).

Another well-known possibility consists in applying *type classes* to represent algebraic structures. Type classes [WB89] are an extension of the functional programming language Haskell enabling *ad-hoc* polymorphism and overloading of operations. Actually, implementing a hierarchy of algebraic structures by means of type classes turns out to be a typical introductory example of their usage [Haf07b]. They are implemented in Isabelle in a way that permits uniformly managing the operational part (as in Haskell) and the logical part.

A possible representation of semigroups as a type class is the following:

```
class times = type +
  fixes times :: 'a => 'a => 'a (infixl * 70)

class semigroup_mult = times +
  assumes mult_assoc: (x * y) * z = x * (y * z)
```

First a type class `times` has been defined containing a single binary operation `times`; then a new type class `semigroup_mult` is derived from it by adding an axiom stating associativity (`mult_assoc`).

A remarkable difference should be pointed out with the representation by means of extensible records. Here the carrier set is not given explicitly. The elements belonging to the algebraic structure (the semigroup) are all the elements belonging to a type satisfying the previous type class specification. An additional drawback of type classes will be relevant in our work. In their Isabelle implementation (and also in the Haskell 98 standard), only type classes with a single parameter (such as `'a` in the previous `times` type class definition) are admitted.

The reason to introduce these two different representations is that both of them were used in our work. We made use of *extensible records* and their properties in our formal proof of the BPL. They allowed us to use the same representation of algebraic structures for basic algebraic structures in our setting but also for algebraic structures containing homomorphisms and endomorphisms. They also permitted us to manage domains and subdomains of algebraic structures in a natural way. On the other hand, the representation of domains by means of sets is not specially suitable with respect to the Isabelle code generation facilities (the main difficulty is that the `set` constructor is too general, since it covers *any* set defined by a predicate; so, without further precisions, generation tools cannot work with our record structures).

From the expressiveness point of view, the type classes approach is roughly equivalent to deal with records without carrier sets (based just on a generic type α). The advantage of using type classes is that the application of the Isabelle code generation facility [Haf07a] to them is straightforward. This is due to a good conceptual linkage between Isabelle type classes and the corresponding Haskell notion. Type classes also provide a mechanism to define *instances* of such type classes, a feature that we introduce in Sect. 4.3 and that we will use in our work. Consequently, in our code generation experiments type classes are the preferred option.

In Sect. 4, we present how these two different representations fit into the code generation machinery.

3.2. Series and nilpotency

Together with the representation of algebraic structures, the representation of the series ϕ and ψ appearing in the BPL statement (Theorem 2.7) and the local nilpotency condition (Definition 2.5) will be our main concern in the code generation process.

The standard Isabelle distribution offers some facilities to represent *sums (or products) of series* (depending on additive or multiplicative notation) which we applied in our formal proof of the BPL to represent the series ϕ and ψ.[1] In this representation, power series admit a representation as finite products over a given (finite) set. The internal implementation relies on certain folding operations over the provided set. The code generation facility cannot be directly applied to this generic representation (generic sets are sometimes not amenable to a computational treatment).

A function f over a domain A satisfies the *local nilpotency condition* if it satisfies that for all $x \in A$, there exists a natural number n such that $f^n(x) = 0$. The local nilpotency condition ensures the *existence* of a bound, which is used to compute the previous (finite) series $\phi(x)$ and $\psi(x)$. On the other hand, the property does not provide an explicit way to compute the bound.

The specification of the bound in our formal proof of the BPL was made by means of the `Least` operator, a restricted version of the Isabelle implementation of Hilbert's ϵ operator to domains with an explicit ordering (for instance, the natural numbers). The code generation facility cannot be applied to this specification.

3.3. Code generation tools in Isabelle/HOL

Isabelle/HOL provides two different tools for code generation. The first, by Berghofer [Ber03b], can extract code from an executable fragment of HOL specifications and also from proofs. The tool supports inductive datatypes, recursive functions (both primitive and well-founded recursion) and also inductively defined relations (predicates).

The executable fragment of HOL formulae essentially precludes existential quantification and Hilbert's choice operator. Most of the basic types in Isabelle/HOL and their associated operations are in fact executable; this includes natural numbers, integers and lists. An example of an operation that is not executable is the `Least` operator, which for a given predicate returns the least natural number satisfying that predicate.

The second tool, based on experiences with the first, was developed by Haftmann and Nipkow [HN07] with the motivation of providing proper support for type classes, which are available in Isabelle/HOL. The tool supports several target languages, currently SML and Haskell, and provides a means of generating code for type instances of a class. Hence the target language needs not necessarily to provide type classes.

Haftmann and Nipkow's tool is restricted to code generation from formulae. It does not support inductive definitions. But since we did not make use of the latter, while type classes turned out to be useful, we chose the second tool for the present work.

4. Different approaches to code generation

In this section, we present different methodologies for code generation from specifications that can be used in Isabelle. The first methodology (presented in Sect. 4.1) pays attention only to the specification of the algorithms being explored and (when possible) generates programs from them. The second methodology (Sect. 4.2) generates algorithms in a similar way to the previous one, and additionally certifies that the inputs of such algorithms satisfy certain specifications. The link between the algorithms generated and the certification of the inputs is not internally encoded. The third methodology (Sect. 4.3) generates algorithms also from given function specifications, and, by means of the *instance* mechanism of type classes, ensures that the types over which algorithms are applied really satisfy the required specifications, generating code also from that types. Finally, (Sect. 4.4) we present a tool that allows direct code execution (but not code generation) in ML from the Isabelle environment, enhancing some of the properties of the previous approaches.

We give a detailed presentation of these methodologies with an example about the composition of homomorphisms, based on the following elementary result:

Lemma 4.1 *Given f and g endomorphisms of a given semigroup S, $f \circ g$ is also an endomorphism of S.*

4.1. Code generation from specifications

In order to get a certified algorithm from the previous mathematical result, we must first introduce some preliminary definitions in Isabelle. Semigroups will be represented by means of records (as illustrated in Sect. 5.1).

[1] As we will later detail, this is not the only option to represent series in the proof assistant.

A homomorphism between two semigroups is a term with a functional type such that it maps elements in the carrier set of the first semigroup to elements of the carrier set of the second semigroup and it preserves the binary operation of the semigroups.

Then, the composition of endomorphisms can be defined as follows:

```
definition composition :: "('a ⇒'a) ⇒ ('a ⇒'a) ⇒'a ⇒'a"
   where "composition g f = g ∘ f"
```

From two given endomorphisms `f` and `g` the composition is defined by means of the ∘ operation over functions (restricted to a single type variable). The code generation tool can be then applied to this definition. The result is an algorithm in a programming language (with the Isabelle code generation facility, we can choose among ML, Haskell or Ocaml code) which takes two arguments (functions in this case) and produces a function (the composition of the two inputs). In the sequel, we will use ML, since the code obtained can be directly executed from the Isabelle environment.

In order to certify that the obtained algorithm really satisfies the property stated in Lemma 4.1, this has to be formally proved in Isabelle:

```
lemma comp_closed: assumes "semigroup S" and "f ∈ hom S S" and "g ∈ hom S S"
    shows "g ∘ f ∈ hom S S"
```

This proof acts as a certificate of the algorithm obtained previously. It ensures that, whenever two endomorphisms are composed, the result of their composition is again an endomorphism.

We chose this methodology in our first experiments about code generation from the BPL [ABR05]. The BPL statement already gives an explicit definition of the objects that have to be computed; it can be said that the output of the BPL algorithm is already present in the BPL statement. Thus, we found this methodology suitable for our experiments.

These first experiments were developed with the code generation facility developed by Berghofer [Ber03a]. Similar results are obtained applying the Isabelle code generation facility by Haftmann and Nipkow [HN07].

4.2. Code generation plus certification of the input

The previous certified algorithm cannot be considered fully satisfactory from a certain point of view. Three explicit assumptions were made in the lemma statement (namely, that `S` is a semigroup, and that both `f` and `g` are endomorphisms of `S`). If we omit these premises and we apply the certified program over a faulty input, for instance, considering the ML type `int` with `+` as the semigroup `S` and `fn x => x + 1` as both endomorphisms `f` and `g`, the obtained ML function will not satisfy the specification of endomorphisms, *since its inputs are not endomorphisms*. In other words, the correct application of the (certified) program is not ensured in our certification context.

In the previous case, we can prove in Isabelle some auxiliary lemmas stating the given assumptions:

```
definition int_semigroup ::"int semigroup"
   where "int_semigroup= (|carrier =(UNIV::int set), mult = op +|)"

lemma int_semigroup_semigroup: shows "semigroup int_semigroup"
```

First we have defined the set of all elements of `int` type in Isabelle with addition (`+`) as a `semigroup` record; then this record is proved to satisfy the locale `semigroup` specification.

Accordingly, we can define a function over the `int` type and prove a lemma stating that such a function is indeed an endomorphism over the semigroup previously defined. For instance:

```
definition double :: "int ⇒ int" where "double = (λx::int. 2 * x)"

lemma double_hom: shows "double ∈ hom int_semigroup int_semigroup"
```

Code generation can be applied to the `double` function, obtaining a function in ML. Finally, the obtained ML function can be composed with itself by means of the ML function `composition` generated in Sect. 4.1. With the lemmas `double_hom` and `int_semigroup_semigroup` we have certified that the input data of our certified

algorithm `composition` satisfies its corresponding specification. In other words, we have explicitly proved every premise in the given lemma `comp_closed` (corresponding to Lemma 4.1).

4.3. Code generation plus type classes and instances

The previous methodology can be made internal to Isabelle by means of *type classes* and *instances*. Details about the implementation of type classes in Isabelle are provided by Haftmann [Haf07b]. We recover now the `semigroup_mult` type class that we introduced in Sect. 3.1.

This type class represents every type satisfying the specified properties (i.e., having a `mult`, or $\otimes$ in infix notation, binary operation and such that it is associative). The type class includes in its definition a single parameter type `'a`, that can be *instantiated* with further types. Giving an instance of this type class requires proving that the proposed type contains similar operational and logical parts:

```
instance int :: semigroup_mult "x ⊗ y ≡ x + y"
proof
...
qed
```

In this example, the operational part is explicitly given to the system, and then the logical part requires an Isabelle explicit `proof`. Following our example about endomorphisms, we can *augment* the type class definition of `semigroup_mult` with additional parameters `f` and `g` which represent two endomorphisms over the semigroup defined:

```
class semigroup_mult_end = semigroup_mult +
  fixes f :: "'a ⇒' a" and g :: "'a ⇒' a"
  assumes "∀x y.f (x ⊗ y) = f x ⊗ f y"
  and "∀x y.g (x ⊗ y) = g x ⊗ g y"
```

Within the previous type class it can be proved that the composition of `f` with `g` produces an endomorphism. If we compare this type class definition with the statement of Lemma 4.1 (or with the Isabelle lemma `comp_closed`) presented in this section, it can be observed that we have exactly augmented the type class specification with the premises of the original lemma. The reason for creating this rather artificial type class is that neither Berghofer's nor Haftmann's code generation facility admits rules with explicit premises, but only equations (or definitional theorems). Therefore, we decided to introduce premises as part of the specification of the type classes being built.

Then, a suitable instance of this type class can be defined starting from the previous `int` instance of class `semigroup_mult` and adding two endomorphisms (both equal to $\lambda\, x.2 * x$) with the properties shown in `semigroup_mult_end`:

```
instance int::semigroup_mult_end "f ≡(λx.2 * x)" "g ≡(λx.2 * x)"
proof
...
qed
```

When proving this instance, the premises in Lemma 4.1 are verified. Then, the code generation process can be summarized as follows. If the code generation facility is applied to a (parameterized) type class definition, type definitions in the target language are obtained. Since Isabelle type classes and instances are based on these Haskell features, their code generation to this programming language is natural. If ML code is preferred, code generation is supported by an explicit *dictionary* construction [Haf07a, HN07]. If code generation is applied to the instances being built (where the type parameter has become an *explicit* Isabelle type), then the code generation tool is applied to this type parameter, and the instance specification is also generated.

In our previous example, generating code from the `int` instance of the type class `semigroup_mult_end` will first generate an ML type based on the Isabelle type `int`,[2] and then, will generate code from the remaining part of the operational specification of the type class (`mult`, `f` and `g`). On the other hand, if a type without an executable counterpart is proved to be an instance of a type class, code generation from this instance will fail.

[2] There already exist facilities to couple common Isabelle types such as `int` or `nat` with their equivalent ones in the target languages.

We now define a function composing the endomorphisms `f` and `g` represented in the type class `semigroup_mult_end`.

```
definition comp_over_end :: "'a::semigroup_mult_end ⇒ 'a"
  where "comp_over_end = (f ∘ g)"
```

If we now apply code generation, after the process explained above we will have (apart from the underlying machinery behind the scenes) an ML function `comp_over_end` that can be executed from the Isabelle environment:

```
ML "comp_over_end (semigroup_mult_end_int) 78"
```

The result of evaluating the previous command in ML is (312 : `int`). The ML's type checker itself will prevent us of executing the previous command, if we have not satisfied the proof obligations derived from the previous *instance* declaration. The system keeps track of all defined type classes and their proven instances, avoiding unsound declarations/computations. Once a type defined in Isabelle has been proven to be an instance of some type class, the type system of the preferred programming language for code generation (SML, Haskell) will be in charge of checking that a generated function or algorithm is only applied to arguments of the type generated from the Isabelle one. Thus, a client of the generated programs does not have to pay attention to the adequacy of the inputs provided to the algorithms. Once the proving effort has been carried out in Isabelle, the sound use of the SML program is ensured by the type system.

Therefore, the whole calculation has been certified. This should be compared with the initial approach in Sect. 4.1, where the program was certified but not the application of the program to correct inputs. We consider that this step from *certified programs* to *certified computations* is a remarkable improvement in our field of application (i.e., homological algebra) where inputs of programs are also complex structures which must satisfy some non-trivial properties (specially, the termination of computations).

4.4. Code execution

There are still other means in the Isabelle environment to execute formal specifications, which do not involve code generation. The best known of them is Isabelle's simplifier, which applies higher-order rewriting based on the available theorems, generating thus new theorems. Unfortunately, its performance is not satisfactory for a computational tool.

The HCL [Obu08] (HOL Computing Library) is another way of obtaining computations from Isabelle executable specifications and theorems. Its original purpose was the computation of the basic linear programs associated to the graphs appearing in the Hales' Flyspeck project to formalize the proof of Kepler's Conjecture [Hal]. HCL is an extension of Isabelle which computations are carried out in SML (this means that such computations are not carried out by the Isabelle inference kernel, but from the SML obtained from the applicable rules). The kind of computations it can accomplish include making use of constant definitions, functions, rewrite rules and also of conditional rewrite rules. It is less powerful than the simplifier, since it does not include congruence rules. Regarding our work, it allows the execution of conditional rewrite rules (that will appear in one of our examples, and were not amenable to code generation with Haftmann's or Berghofer's tools), and its performance is various orders of magnitude faster than the one of the simplifier. For instance, our previous example about the composition of two endomorphisms can be adapted to this tool by simply facilitating the HCL the definitions to be used (`f` and `g` over the integers, `comp_over_end`) and some auxiliary results enabling computations, and the tool will compute the corresponding results.

In Sect. 7.2, we will present two different representations of matrices. The results linking both representations (defining the domain transformation) will include conditional equations, and thus direct code generation from them will fail. In that situation, we will apply the HCL, that will help us to carry out computations over the first representation by means of the second one.

5. Adapting the formal proof

In this section, we present the modifications that had to be done in the formal proof of the BPL to accomplish the restrictions derived from adjusting it to the code generation ideas introduced in the previous section. The goal is

to find an alternative representation for some of the solutions used in our formal proof of the BPL (presented in Sect. 3), which can be later used with the code generation facility, and in such a way that the prior formal proof of the BPL can be applied with the minimum effort (or changes).

Two options can be considered to this end. The first one would consist in implementing a *complete formal proof* of the BPL in Isabelle/HOL, where the specification of algebraic structures (Sect. 3.1) and series (Sect. 3.2) would be adequately modified, and our first formalized proof used as a model. This option is likely to be unfeasible: The formal proof of the BPL presented in [ABR08] was based, roughly speaking, on the idea of defining a *subset (or subdomain)* of one of the differential groups, and proving it isomorphic to another differential group. The explicit representation of domains permits their substitution by proper subsets in a direct way [ABR04, GMR07]. If algebraic structures are specified by means of type classes, the idea of representing simultaneously domains and subdomains is no longer possible. Without explicit carrier sets, and in the abscence of subtyping, a subdomain would be of a different type, and explicit transformation between the types would be required to relate them.

The second option consists in *directly reusing* the previous Isabelle proof. This idea takes advantage of the fact that in the BPL statement (see Theorem 2.7) no explicit subsets appear, which means that the statement can be represented by means of type classes. In Sect. 5.1, we present a way to link type classes with the extensible records appearing in the old proof. In Sect. 5.2, we present a computable version of the series in the BPL statement and of the *local nilpotency condition* (such that the code generation facility is applicable to them) and prove them equivalent to the specifications presented in Sect. 3.2. With these results, the formal proof of the BPL can be transferred to the new setting.

5.1. Representing algebraic structures

The problem of linking type classes and records is solved by means of a family of Isabelle *functions* which from a given type class produces a corresponding record. These functions have to be proved to preserve the locale specification linked to the record types, making use of the logical part of the type classes specification.

As an example, for the previously given Isabelle specifications of semigroup as a record type plus a locale specification and as a type class (Sect. 3.1), the following function is defined:

```
definition semigroup_functor :: "('a ⇒ 'a ⇒ 'a) ⇒ ('a semigroup)"
  where "semigroup_functor prod = (|carrier = UNIV, mult = prod|)"
```

We have named the function definition `semigroup_functor` and it can be read as follows: From a given *binary operation* `prod` we build a *semigroup record*, whose carrier set is the set of all elements of the underlying type (`UNIV`) and whose binary operation is `prod`. This definition also helps to illustrate the difference between type classes and the implementation of algebraic structures with records where the carrier set is *explicit*.

The previous definition must be proved to preserve the locale specification `semigroup`. From a given binary operation `prod` satisfying the `semigroup_mult` type class specification (i.e., associativity), the record being built satisfies the specification of the locale `semigroup` (i.e., having a binary operation that is closed over the given carrier and associative):

```
lemma assumes "semigroup_mult prod" shows "semigroup (semigroup_functor prod)"
```

Similar results are proved for the different kinds of algebraic structures in our setting (monoids, groups, abelian groups, differential groups), allowing us to build a link among algebraic structures represented as type classes and their specification by means of record types. Accordingly, functions representing homomorphisms between type classes had to be transformed into functions between record types. It can be said that the new proposed representation of algebraic structures and homomorphisms (by means of type classes and functions among them) is *formally proved* to be a particular case of the representation in the Isabelle formal proof of the BPL.

A major limitation of the Isabelle implementation of type classes has been already pointed out: they only admit a *single* type parameter. The BPL statement cannot be represented as a single parameter type class (its natural representation requires different types for the different differential groups involved). Due to this limitation, we made use of two different type classes: One of them to represent the differential group (C, d_C) in Theorem 2.7, and another one to produce an *augmented* representation of (D, d_D).

What we mean by augmented is that, taking advantage of the fact that both the homotopy operator (h) and the perturbation (δ_D) appearing in the BPL statement are *endomorphisms* of (D, d_D), they can be implemented as operations in the (single parameter) type class representing (D, d_D); we can also introduce in this type class *all premises* in the BPL statement involving such endomorphisms (following the idea suggested in Sect. 4.3 for the simpler case of semigroup endomorphisms). These premises include that both h and δ_D are endomorphisms of the differential group, that h is such that $hh = 0_{\mathrm{End}\, D}$, that δ_D is such that $d_D + \delta_D$ is again a differential of D and, finally, that δ_D verifies the local nilpotency condition (with respect to h).

This type class together with the instance mechanism will help to certify the correctness of the inputs applied to the algorithms generated.

5.2. Computing the local nilpotency

In this section, we present a way to overcome the difficulties pointed out in Sect. 3.2 to obtain a computable version of the series ϕ and ψ appearing in the BPL statement (defined in Theorem 2.7). The *code generation facility* has to be applicable to these series, and, of course, they have to be proved equivalent to the specification of ϕ and ψ in the original formal proof (as presented in Sect. 3.2).

The first difficulty was due to the definition of the finite sum (or product) of the terms of a series. The generic implementation over generic sets used in our formal proof (to which the code generation tool was not directly applicable) is here overcome by proposing a new definition restricted to the (inductively defined) set of the natural numbers. This restricted version is implemented by means of primitive recursion, and generating code from it is straightforward:

```
primrec
"fin_sum f 0 = id"
"fin_sum f (Suc n) = f^(Suc n)+(fin_sum f n)"
```

The operation + denotes the pointwise addition over the underlying domain.

The second difficulty originated from the local nilpotency condition. We first introduce a generic Isabelle predicate to determine locally bounded functions (in particular, due to the local nilpotency premise, $\delta_D h$ will satisfy this predicate):

```
constdefs
  "local_bounded_func (f::'a ⇒ 'a) == (∀ x. ∃ n. (f^n) x = 0)"
```

The next step is to define an *algorithm* computing the bound `n` in the previous definition. Here, we borrow ideas due to Obua [Obu07]. There, an implementation of partial functions that can be represented by *while* and *for* loops is suggested in the HOL setting, based on the *While* and *For* combinators (which can be defined as tail recursive functions, accepted by the *function* package by Krauss [Kra07]). In that work, these combinators are used for the computation of *orbits* of functions over a given point. In some sense, the computation of the consecutive powers of $\delta_D h$ over some x until it vanishes is reminiscent of the computation of the orbit of a function at a given point, and we were able to apply similar techniques.

As a first solution to the problem of defining a function that computes the bound of nilpotency of a given function `f` over an element of its domain `x`, the following Isabelle definition of a tail recursive function succeeds:

```
function (tailrec) bound ::"('a ⇒ 'a) ⇒ 'a ⇒ int ⇒ int"
   where "bound f x n = (if (x = 0) then n else bound f (f x) (n + 1))"
```

Some comments are worth noting on the previous definition. First, the `bound` function has been defined making use of the Isabelle *function* package [Kra07]. This package eases the definition of recursive functions in Isabelle and is implemented as a definitional extension of the system. In the previous definition, the facilities provided have been used to define `bound` as a tail recursive function.

Tail recursive functions always have a total model, even if they are non-terminating (for instance, in our previous definition, it might be possible that `f` never vanishes and then an infinite loop would arise). A further explanation of the ideas behind is given by Owens and Slind [OS08]. The possibility to define `bound` as a tail recursive function is crucial for our intentions of generating code from it. Usually, partial function definitions give rise to an internal definition in Isabelle which contains some partiality premises. The defining rules obtained are incompatible with the code generation facility, which can be only applied to *equations*. Since tail recursive

functions have total models, their defining rules can be seen as equations and they can be used to generate *code* from.[3]

Instead of using the previous `bound` definition directly, we preferred making use of the *For* and *While* combinators. The function `local_bound_gen` below (aimed at substituting the previous `bound` function definition) computes the bound (`n`) of a function `f` in `x` and is defined in our setting as the following `For` combinator:

constdefs
"local_bound_gen f x n == For (λy.y ≠ 0) f (λn. n+1) x n"

The previous *For* combinator can be defined directly in terms of the *While* combinator, and once again as a tail recursive function from which we can take advantage for our proof obligations. This definition, together with the termination condition provided by the `local_bounded_func` predicate previously defined, gives the user a simpler induction rule for *For* combinators (than the one automatically provided by the *function* package for tail recursive functions), which permitted us to prove the following result in Isabelle:

lemma assumes *"terminates (λy.y ≠ 0,f,x)"* **shows** *"local_bound_gen f x 0 = (LEAST n.(f^n) x = 0)"*

This Isabelle result states the equivalence of the specification of the *bound* of local nilpotent series used in the formal proof of the BPL (as presented in Sect. 3.2), and the specification in terms of the function `local_bound_gen` (which is admitted as input for the code generator), under a suitable premise of termination (given with the `assumes` command), which in our setting is ensured by the *local nilpotency condition*.

Therefore, our two requirements have been satisfied. First, we have produced an algorithm computing the value of a power series satisfying a local nilpotency premise over a given point, and second, we have proved the equivalence between this value and the corresponding one in the old formalization in our proof of the BPL.

6. Code generation from the formal proof

In this section, we present the ML code obtained from the specifications presented in the previous section. In Sect. 6.1, we introduce the code obtained from the hierarchy of algebraic structures specified by means of type classes. In Sect. 6.2, we present the code generated from the series as presented in Sect. 5.2.

6.1. Code generation from the algebraic structures

In Sect. 5.1, we have mentioned a hierarchy of type classes used to represent the algebraic structures appearing in the BPL proof. The code generation facility of Isabelle is now to be applied to these type classes. The code generated from a type class definition is a (parameterized) type definition in ML. Let us illustrate the code generated from two type classes `semigroup_mult` and `monoid_mult`:

```
type 'a semigroup_mult = {times: 'a -> 'a -> 'a};

type 'a monoid_mult =
  {monoid_mult_semigroup_mult: 'a semigroup_mult, one: 'a};
```

Operations in the original type classes preserve their arities, and types can be augmented with additional fields to get more elaborated ones. In this way we obtain ML type definitions for the collection of type classes evoked in Sect. 5.1. The lack of type classes in ML is solved by means of explicit dictionary constructions [Haf07a, HN07].

6.2. Code generation from the series

As we have introduced in Sect. 5.2, the function `local_bound_gen` allows the computation of the bound of the series appearing in the BPL statement. The code generation process applied to this function produces the following ML code (we have omitted long identifiers and some explicit casts between the types involved):

[3] Quoting from the Isabelle *function* package manual [Kra07]: "Thus, in order to generate code for partial functions, they must be defined as a tail recursion".

```
fun whilea continue f s =
    (if continue s then whilea continue f (f s) else s);

fun for' continue f aca x ac =
  whilea (fn a as (acb, aa) => continue aa)
    (fn a as (acb, xa) => (aca xa acb, f xa)) (ac, x);

fun for continue f aca x ac =
    Product_Type.fst (for' continue f aca x ac);

fun local_bound_gen (A1_, A2_) f x n =
   for
    (fn y => not (op_eq A1_ y (zero A2_))) f
        (fn y => fn na => plus_nat na (Suc Zero_nat)) x n;
```

First, it should be noted again that the code generation process applied to function `local_bound_gen` has succeeded since the function has been encoded as a tail recursive function (by means of the `For` operator). Encoding it as a partial recursive function would make the code generation process fail.

The code generated from the function gives rise to a tail recursive function, based on a restricted version of a *while* loop (this may be seen in the definition of function `whilea`). The *continue* parameter is a predicate expressing the condition ending the iterative process (in our case, the code generated for the function $(\lambda\, y.y \neq 0)$). Then, the loop produces iteratively the n-th power of f applied to a given x.

The function `local_bound_gen` will produce the bound of nilpotency of a series (if the general term of the series satisfies the nilpotency condition). The remaining part of the computation of the series is expressed (and generated in ML) as the finite sum (from the definition of `fin_sum` introduced in Sect. 5.2) of the consecutive powers (`fun_pow`) of a function `f` (again we have omitted some type casts and long identifiers). The auxiliary function `plus_fun` implements the point-wise addition of two functions `f` and `g` for a given `x`:

```
fun plus_fun A_ B_ f g x = plus B_ (f x) (g x);

fun fun_pow A_ f n =
  (if ((n:int) = (0:int)) then id
    else f o fun_pow A_ f (nat (- (n, (1:int)))));

fun fin_sum A_ f n =
  (if ((n:int) = (0:int)) then id
    else plus_fun A_ A_
           (fun_pow A_ f (+ (nat (- (n, (1:int))), 1)))
           (fin_sum A_ f (nat (- (n, (1:int))))));
```

Finally, when we generate code from the Isabelle definition of the series `phi` and `psi`, the definitions of `fin_sum`, the opposite of the composition of `pert` and `h`, and the previous function `local_bound_gen`, are brought together. It is worth noting that both functions `phi` and `psi` are defined over the ML type obtained from the *augmented* type class defined for differential groups containing a perturbation, a homotopy operator, and satisfying the local nilpotency condition. This implies that, prior to computing the value of `phi` over a given instance, we must first prove that the instance satisfies this specification. We explain the details of this process in Sect. 7.

7. Examples of execution

The programs obtained in the previous sections can be now used in two different ways. The first one consists in providing suitable ML inputs for these programs and executing them in any system containing an ML compiler. This idea was proposed in Sect. 4.1. The second way consists in providing (and thus, proving) in Isabelle suitable *instances* of the type classes being used, and then running the ML programs over the ML types generated from such instances. This technique was proposed in Sect. 4.3. The first technique is straightforward and does not require further explanations. In this section, we present two examples of application of the second technique.

They are two different examples of the BPL. The first one (Sect. 7.1) presents a reduction from a differential group $(\mathbb{Z}^4, d)$ to another one $(\mathbb{Z}^2, d')$. The second one (Sect. 7.2) introduces a reduction from a bicomplex to a null differential group.

7.1. An example with global nilpotency

Let us consider the differential groups $D = \mathbb{Z}^4$ with differential $d_D(a, b, c, d) = (0, 2a, 0, c)$ and $C = \mathbb{Z}^2$ with differential $d_C(a, b) = (0, 2a)$. The homotopy operator for D will be $h(a, b, c, d) = (0, 0, d, 0)$ and $\delta_D(a, b, c, d) = (0, a + c, 0, a)$ the perturbation. Finally, the homomorphisms f and g between both differential groups are given by $f(a, b, c, d) = (a, b)$ and $g(a, b) = (a, b, 0, 0)$.

The first step will be to choose a representation for this example in Isabelle. The HOL type system includes the product of types as a type constructor. Making use of tuple types, we gave representations for $C = \mathbb{Z}^2$ and $D = \mathbb{Z}^4$. The operations in the previous paragraph were also defined. The types and operations defined have to be proved to be *instances* of the type class specified. The representation by means of tuples of $C = \mathbb{Z}^2$ with the operation $d_C(a, b) = (0, 2a)$ was proved to be an instance of the type class representing differential groups. Accordingly, the representation chosen for $D = \mathbb{Z}^4$ with differential $d_D(a, b, c, d) = (0, 2a, 0, c)$, $h(a, b, c, d) = (0, 0, d, 0)$ as homotopy operator and $\delta_D(a, b, c, d) = (0, a + c, 0, a)$ as perturbation was proved to be a differential group, with h and δ_D being a homotopy operator and a perturbation, and δ_D satisfying the local nilpotency condition with respect to h. The last fact ensures that the previous generated ML function `local_bound_gen` will not infinitely loop whenever it is applied over an element of the ML type generated from the Isabelle tuples representing $\mathbb{Z}^4$.

It may be mentioned that the local nilpotency condition in this concrete example becomes a *total* nilpotency condition, since $n = 2$ makes the composition $(-\delta_D h)^n$ vanish for every element in $\mathbb{Z}^4$. Nevertheless, this property does not pose a difference from the internals of the proofs carried out.

After proving the instance, the code generation process produces ML programs from the given specifications. For instance, the ML types representing pairs and four tuples are:

```
datatype 'a sProd = SPair of 'a * 'a;

datatype 'a quad_type_const = Quad of 'a sProd * 'a sProd;
```

The type parameter `'a` will be later instantiated with the ML type `int`. The previous operations are then generated in a similar fashion. They resemble the ML code that we presented in Sect. 6, just taking into account the more accurate specifications of the operations defined (for instance, the differential over $\mathbb{Z}^4$). On the other hand, the definitions of `phi` and `psi`, being defined as constants over the type class, are not generated again, but just reused as the ones we presented in Sect. 6.2.

Some examples of execution of the previous program follow. We define them as constants in the Isabelle setting from which we generate code. For instance, the following Isabelle definition applies the series ϕ to the four-tuple (5, 3, 8, 9). The type hint `int` is enough for the code generator to identify the concrete instance we are referring to.

```
definition "foos == Φ((5::int),3,8,9)"
```

The code generator is then applied to the previous constant and it can be executed directly from the Isabelle environment by means of the command:

```
ML foos
```

The obtained result is the tuple $(5, -6, 8, 9)$.

Not only the result of computing the series ϕ and ψ can be obtained in our setting, but also the results of applying the different components of the reduction to any given input (for instance, the result of applying `f'` over a given tuple). Nevertheless, in this last case, a subtle difference must be pointed out. As we already mentioned in Sect. 5.1, type classes only admit a single type parameter. As `f'` (or similarly `g'`) is a function between different types, it cannot be included in any type class in our setting, and, consequently, has not been proved to satisfy its specification as stated in the BPL.

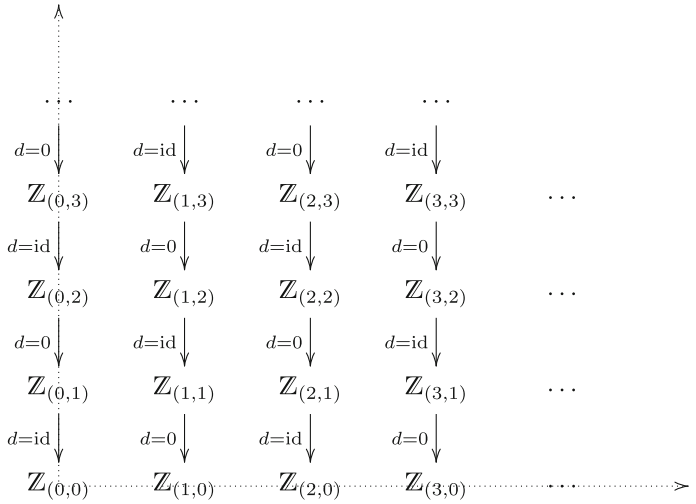

Fig. 1. Definition of the differential d of a bicomplex

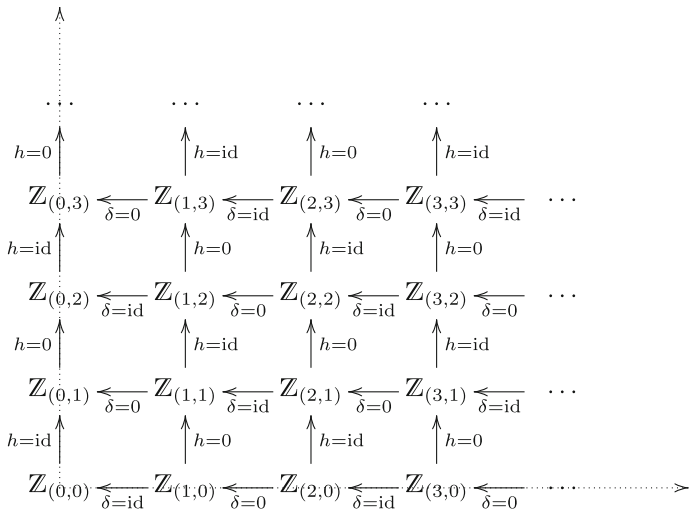

Fig. 2. Definition of the homotopy operator h and the perturbation δ of a bicomplex

7.2. An example with local nilpotency

A second example of the BPL is the following. Let D be a bicomplex, and C the null differential group. A bicomplex (including its differential)[4] can be depicted as shown in Fig. 1. The components of the bicomplex can be seen as linear combinations of integer numbers containing their coordinates. For instance, $((3, (0, 0)), (2, (1, 0)), (-5, (0, 1)))$ would be an element of the bicomplex, represented as a list of pairs where the first element is an integer and the second element of the pairs is the coordinate. The basic operations (equality, addition) over the bicomplex are defined componentwise. The differential given satisfies the requirement $dd = 0$.

The homotopy operator h and the perturbation (denoted by δ) are represented in Fig. 2. The operators h and δ (as shown in Fig. 2) satisfy the requirements in the BPL statement. For instance, it can be easily seen from the graphic that $hh = 0$. The local nilpotency condition of δ can also be proved following the paths of the composition δh along the graphic.

[4] The differential is denoted by d instead of d_D for the sake of readability.

A reduction between the bicomplex and the null differential group is then given by the null homomorphisms $f = 0$ and $g = 0$.

Different representations can be proposed from the previous algebraic structure. For instance, matrices with integer elements. Two different representations of matrices are available in the Isabelle standard distribution [Obu05] (this work is part of the effort of providing a formalization of Kepler's Conjecture [Hal]):

- The first representation of matrices is based on the following type definition:

```
types 'a infmatrix = "[nat, nat]⇒'a"

definition nonzero_positions :: "('a::zero) infmatrix ⇒ (nat*nat) set"
  where "nonzero_positions A = {pos. A (fst pos)(snd pos) ≠ 0 }

typedef 'a matrix = "{f::(('a::zero) infmatrix). finite (nonzero_positions f)}
```

  This type is isomorphic to the set of functions of type `nat * nat => 'a` that map only a finite number of points to nonzero values.

- The second representation defines *sparse* matrices as follows:

```
types
'a spvec = "( nat * 'a) list"
'a spmat = "('a spvec) spvec"
```

  Every row is represented as a list of pairs of type `nat * 'a`, and then a matrix as a list of pairs formed by natural numbers and rows. Thus, null elements of a matrix can be omitted from its representation.

The first one has been designed to develop proofs, due to its favorable algebraic properties. For instance, every matrix has a unique representation in that setting. Operations can be defined in a very natural way over this representation and consequently proofs also become rather simple in this structure.

We were able to prove in Isabelle this type definition an instance of the type class representing differential groups. We also proved that the homotopy operator h and the perturbation δ satisfy the requirements found in the BPL. Finally, we proved that δ satisfies the local nilpotency condition. In order to prove such a property we applied an auxiliary lemma stating that there is no infinite decreasing sequence of *natural* numbers (just as there is no infinite decreasing path in Fig. 2).

While the properties of this matrix representation facilitate the development of proofs, it poses a limitation when it comes to code generation. This generic type does not have a computational counterpart, which means that, once we have proved it to be an *instance* of the type class, code generation from it failed.[5]

In order to obtain an executable representation of matrices and their operations, Obua proposed the second representation shown above. In this setting, definitions for usual operations such as addition and multiplication over matrices are provided. Furthermore, we provided the definitions of the differential, the homotopy operator and the perturbation required in our example. Both the matrices representation (i.e., its type definition) and the operations (implemented by means of recursive functions over the matrices) were amenable to code generation in Isabelle, but proving properties about them was rather complicated (and had been already done over the first representation). For instance, some of the properties to be proven required additional premises about the elements of matrices being sorted.

Accordingly, we defined a *domain transformation* between the first representation of matrices and the second one. We had to prove lemmas stating that applying operations (addition, differential, homotopy operator, or combinations of them, up to the series ϕ and ψ in Theorem 2.7) over the first representation (that had been formally proved to be an instance of the input of the BPL algorithm) produced the same result as applying some new operations defined over the second representation. These lemmas have the form of conditional rewrite rules, where the premises usually concern the sortedness of matrices in the second representation. Thus, once again (as in the case of the transformation from type classes to extensible records), we had to use the idea of proving in an adequate setting, and then obtaining executable counterparts, too.

Since the previous lemmas enabling the domain transformation had the form of conditional rewrite rules, we were prevented from using the code generation tools by Berghofer or Haftmann to obtain computations over

[5] This problem was already mentioned in Sect. 4.3 when talking about the algebraic structures representation and the code generation from instances.

matrices. Instead, the HOL Computing Library was applicable in this situation. The HCL also accepts a set of equations and translates them into code, but, in contrast to the code generation tools, the HCL can also deal with conditional equations, where the conditions must of course be executable themselves. In our application, the preconditions are structural conditions on the matrices that are executable.

We recall here that the HCL provides the user with *computations*, but does not generate code. Thus, we were capable of carrying out *certified computations* using the operations defined over the first representation of matrices presented in this section. For instance, given the following matrix definition, (*w.r.t.* the second representation):

```
definition "example ==
( 0::nat, ( 0::nat, 5::int) # ( 2,-9) # (6, 8) # []) #
( 1::nat, ( 1::nat, 4::int) # ( 2, 0) # (8, 8) # []) #
( 2::nat, ( 2::nat, 6::int) # ( 3, 7) # (9,23) # []) #
(16::nat, (13::nat,-8::int) # (19,12) # []) #
(29::nat, ( 3::nat, 5::int) # ( 4, 6) # (7, 8) # []) # []"
```

Now it can be computed in ML the nilpotency bound of $\alpha = -(\delta h)$ for this matrix by means of the following command:

```
ML{*compute @{cterm "local_bound α (sparse_row_matrix example)"}
```

The result obtained from executing the previous command (from the Isabelle environment itself) is:

```
val it = "local_bound α (sparse_row_matrix example) ≡ 30": Thm.thm
```

Note that the computing library returns an equation with the result as the right hand side. The result of the computation is the number of rows, 30 (as might be expected from a closer look to Fig. 2). The syntax of the command itself may shed light on the internal processes that are taking place. First, we are directly invoking the function `local_bound` over α; both α and `local_bound` correspond to the functions defined in our first (and non-executable) representation of matrices to compute the bound of nilpotency (and which have been proved to satisfy the condition of termination over any matrix). These functions, being defined over matrices represented as sets, need to receive as a parameter a matrix of such a type. Thus, we must apply the *domain transformation* (represented by the function `sparse_row_matrix`) converting `example`, which has been defined as a *matrix w.r.t. the second representation*, into a matrix represented as a set.

The HCL presents results of computations in the Isabelle environment as theorems (the above output is of type `Thm.thm`). It is important to note that this is not a theorem derived by the Isabelle inference kernel, since the computation has been carried out in ML. The HCL merely asserts the result of the computation as an axiom.[6]

In a similar way, we can compute the result of evaluating the Isabelle definitions of the series Φ or Ψ over any given matrix.

```
ML{*compute @{cterm "Φ (sparse_row_matrix example)"}
```

The following result will be obtained:

```
val it  = "Φ (sparse_row_matrix example) ≡ sparse_row_matrix
   [( 0, [( 0, 5), ( 2,-13), ( 4, 6), ( 6, 8), ( 32, -5), ( 36, -8)]),
    ( 1, [( 1, 4), ( 3,-6), ( 8,  8), (31,  5), (35, 8)]),
    ( 2, [( 2, 6), ( 3, 7), ( 9, 23), (30, -5), (34,-8)]),
    ( 3, [(29, 5), (33, 8)]),   ( 4, [(28,-5), (32,-8)]),
    ( 5, [(27, 5), (31, 8)]),   ( 6, [(26,-5), (30,-8)]),
    ( 7, [(25, 5), (29, 8)]),   ( 8, [(24,-5), (28,-8)]),
    ( 9, [(23, 5), (27, 8)]),   (10, [(22,-5), (26,-8)]),
    (11, [(21, 5), (25, 8)]),   (12, [(20,-5), (24,-8)]),
    (13, [(19, 5), (23, 8)]),   (14, [(18,-5), (22,-8)]),
    (15, [(17, 5), (21, 8)]),   (16, [(13,-8), (16,-5), (19,12), (20,-8)]),
    (17, [(15, 5), (19, 8)]),   (18, [(14,-5), (18,-8)]),   (19, [(13, 5), (17, 8)]),
    (20, [(12,-5), (16,-8)]),   (21, [(11, 5), (15, 8)]),   (22, [(10,-5), (14,-8)]),
    (23, [(9,  5), (13, 8)]),   (24, [(8, -5), (12,-8)]),   (25, [(7,  5), (11, 8)]),
    (26, [(6, -5), (10,-8)]),   (27, [(5,  5), (9,  8)]),   (28, [(4, -5), (8, -8)]),
    (29, [(3,  5), (4,  6), (7, 8)])]": Thm.thm
```

[6] This feature is not intrinsic. The HCL could have also returned the result as a value. Having the axiom available was desirable in the Flyspeck project, Obua's main application of the HCL.

Note that the result of this computation, like the input, is over the abstract matrix type. The series Φ need not be defined over a type amenable to code generation. Instead, rewrite rules associated to the domain transformation `sparse_row_matrix` enable the HCL to translate the input into an executable representation (of pairs and lists in this instance) in which the actual computation takes place.

Once again, we emphasize that the series Φ has been proved to satisfy the BPL specification. Also the collection of domain transformation lemmas has been proved correct. That is, all the input to the HCL has been formally verified, which means that the computation can be considered a *certified computation*.[7]

8. Conclusions and further work

The source code of the previous formalization and examples is available from [Ara08]. The new formalization and transferring the proof (the files on which Sect. 5 is based upon) required *ca.* 1.500 lines of Isabelle code. The development carried out in the concrete instances of the BPL (Sects. 7.1 and 7.2) required 350 and 4.500 (respectively). In comparison, the formal proof of the BPL in [ABR08] consumed *ca.* 4.500 lines of Isabelle code.

The initial goal of our work consisted in obtaining certified code from the BPL. We started from a mechanized proof of the BPL in Isabelle that did not pay attention to computability matters. The BPL algorithm is based on the computation of a bounded series that, correctly applied to an input reduction, produces a new reduction. In this paper it is proposed a way to compute such bound as a tail recursive function. This function has been defined in Isabelle (by means of the *For* combinator) and proved to be equivalent to our previous definition based on the Isabelle specification of the `Least` operator.

From the previous bound and the proofs showing the equivalence between the setting of our mechanized proof and the new provided definitions, we were able to apply the previously proved theorems. Furthermore, we introduced type classes with the aim of increasing the reliability of our computations. On the other hand, type classes in Isabelle admit only single parameter types, and this imposed a major limitation to the use that we made from them and the advantages we obtained. Even so, they allowed us to represent accurately the algebraic structures (the differential groups and some additional premises) in the BPL.

The *instance* mechanism of type classes allowed us to prove the correctness of the underlying types over which our programs compute. Nevertheless, it also imposed some limitations with respect to the kind of types that we can use (namely, only computable ones). We obtained a non-trivial complete example of execution, presented in Sect. 7.1, that showed the applicability of our ideas.

Then we tackled a more elaborated example in Sect. 7.2, in which some implementation problems were encountered. Proving that matrices were a concrete instance of the type class representing differential groups and the homotopy operator and the perturbation verifying the local nilpotency condition was an already interesting result of the applicability of the created environment. Unfortunately, the code generation tool was not applicable to this representation. Then, we used a different representation of matrices known to be compatible with the code generation tool in its actual state (based on lists of lists, and where the operations we require can be recursively defined). We were capable of linking both representations by means of appropriate lemmas proving the equivalence of application of the different operations. This domain transformation, together with the HCL, enabled the computation of the BPL algorithm over bicomplexes.

The applicability of our methodology depends only on two matters: a type definition of the algebraic structure in question has to be provided and, secondly, types and their operations need to have an executable specification *w.r.t.* the code generation capabilities. The two examples in Sect. 7 show that our ideas can be applied to every case of the BPL where the differential groups are free bicomplexes or tuples over the integers. Additionally, other groups can be encoded as instances of the type classes described; a concrete finite cyclic group, for instance, can be represented by means of an enumerated type, and then proved to satisfy the specification of the type classes appearing in the BPL statement.[8] The second requirement, of having an executable specification, can be relaxed. By providing a domain transformation as introduced in Sect. 7.2 it is sufficient that the type be mapped into another type with an executable specification.

As a natural continuation of this work, the implementation of multi-parameter type classes in Isabelle/HOL could be explored. This kind of type classes are needed to give a type class including all the information presented

[7] We could have equivalently carried out the computation inside of Isabelle, by means of the same equations, obtaining thus a theorem derived by the Isabelle inference kernel, but it would be significantly (several orders of magnitude) slower.

[8] Let us stress that this is different from having a HOL type which represents generically finite cyclic groups, a problem that is not addressed in this paper.

in the BPL statement. Therefore, if we aspire to prove instances of the BPL (and not only of the algebraic structures appearing in it) in Isabelle, from which we can execute programs where the input has been fully verified, we must be capable of using type classes with different (two, in our case) type variables. Some Haskell compilers support them, but the type system then becomes undecidable or unsound (see [JJM97] for details and some partial solutions).

A different possible solution than multi-parameter type classes would consist in using locale interpretation [Bal06], which allows multiple type variables. Its compatibility with the code generation facility and the function package, as well as some flexibility to define and prove instances, would make it a satisfactory solution.

As a conclusion, in this paper we have shown how an example of software certification in the field of computer algebra has led us to apply a wide range of the facilities available in a proof assistant system (Isabelle/HOL). This shows how in the last few years these systems have grown up very rapidly, offering solutions to almost every problem we have found in our development. At the same time, we have proposed some solutions and approaches which might be used in some similar problems.

Acknowledgments

We would like to thank the anonymous referees for their suggestions, which have lead to substantial improvement of the original manuscript. The first author thanks also Prof. Tobias Nipkow and the members of the "Theorem Proving Group" at the Technische Universität München for hosting him during the elaboration of part of this work and patiently answering every question about the Isabelle/HOL environment.

References

[ABR04] Aransay J, Ballarin C, Rubio J (2004) Four approaches to automated reasoning with differential algebraic structures. In: Buchberger B, Campbell JA (eds) AISC 2004, 7th International conference on artificial intelligence and symbolic computation, Linz, Austria, September 2004. Lecture notes in artificial intelligence vol 3249. Springer, Heidelberg, pp 222–235

[ABR05] Aransay J, Ballarin C, Rubio J (2005) Extracting computer algebra programs from statements. In: Moreno-Díaz R, Pichler F, Quesada-Arencibia A (eds) EUROCAST 2005, 10th international conference on computer aided systems theory, Las Palmas de Gran Canaria, Spain, February 2005. Lecture notes in computer science, vol 3643. Springer, Hidelberg, pp 159–168

[ABR08] Aransay J, Ballarin C, Rubio J (2008) A mechanized proof of the Basic Perturbation Lemma. J Autom Reason 40(4):271–292

[Ara06] Aransay J (2006) Mechanized reasoning in homological algebra. PhD thesis, Universidad de La Rioja, http://www.unirioja.es/servicios/sp/tesis/tesis34.shtml

[Ara08] Aransay J (2008) Code generation from the Basic Perturbation Lemma in Isabelle/HOL, http://www.unirioja.es/cu/jearansa/BPL/code_generation/index.html

[Bal04] Ballarin C (2004) Locales and locale expressions in Isabelle/Isar. In: Berardi S, Coppo M, Damiani F (eds) TYPES 2003, 3rd international workshop on types for proofs and programs, Torino, Italy, May 2003. Lecture notes in computer science, vol 3085. Springer, Heidelberg, pp 34–50

[Bal06] Ballarin C (2006) Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein JM, Farmer WM (eds), MKM 2006, 5th international conference on mathematical knowledge management, wokingham, UK, August 2006. Lecture notes in artificial intelligence, vol. 4108. Springer, Heidelberg, pp 31–43

[Ber03a] Berghofer S (2003) Program extraction in simply-typed higher order logic. In: Geuvers H, Wiedijk F (eds) TYPES 2002, 2nd international workshop on types for proofs and programs, Berg en Dal, The Netherlands, April 2002. Lecture Notes in Computer Science, vol 2646. Springer, Heidelberg, pp 21–38

[Ber03b] Berghofer S (2003) Proofs, programs and executable specifications in higher order logic. PhD thesis, Technische Universität München

[BL91] Barnes DW, Lambe LA (1991) Fixed point approach to Homological Perturbation Theory. Proc Am Math Soc 112(3):881–892

[CS07] Coquand T, Spiwack A (2007) Towards constructive homological algebra in type theory. In: Miner R, Kauers M, Kerber M, Windsteiger W (eds) 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 2007. Lecture notes in computer science, vol 4573. Springer, Heidelberg, pp 40–54

[DLR07] Domínguez C, Lambán L, Rubio J (2007) Object-oriented institutions to specify symbolic computation systems. Rairo Theor Inf Appl 41:191–214

[DSS99] Dousson X, Sergeraert F, Siret Y (1999) The Kenzo program. http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/, April 1999

[GMR07] Gonthier G, Mahboubi A, Rideau L, Tassi E, Théry L (2007) A modular formalisation of finite group theory. In: Schneider K, Brandt J (eds) TPHOLs'07, 20th international conference on theorem proving in higher-order logics, Kaiserslautern, Germany, September 2007. Lecture notes in computer science, vol 4732. Springer, Heidelberg, pp 86–101

[Gug72] Gugenheim VKAM (1972) On the chain complex of a fibration. Ill J Math 16(3):398–414

[Haf07a] Haftmann F (2007) Code generation from Isabelle/HOL theories. Technical report, Technische Universität München, http://isabelle.in.tum.de/doc/codegen.pdf

[Haf07b] Haftmann F (2007) Haskell-style type classes with Isabelle/Isar. Technical report, Technische Universität München, http://isabelle.in.tum.de/doc/classes.pdf

[Hal] Hales T, The flyspeck project. http://code.google.com/p/flyspeck/

[HN07] Haftmann F, Nipkow T (2007) A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern

[JJM97] Jones S, Jones M, Meijer E (1997) Type classes: an exploration of the design space. In: Proceedings of the Haskell Workshop, Amsterdam

[KP99] Kammüller F, Paulson LC (1999) A formal proof of Sylow's Theorem—an experiment in Abstract Algebra with Isabelle/HOL. J Autom Reason 23(3):235–264

[Kra07] Krauss A (2007) Defining recursive functions in Isabelle/HOL. http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf

[LPR03] Lambán L, Pascual V, Rubio J (2003) An object-oriented interpretation of the EAT system. Appl Algebra Eng Commun Comput 14(3):187–215

[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher order logic. Lecture notes in computer science, vol 2283. Springer, Heidelberg

[NW98] Naraschewski W, Wenzel M (1998) Object-oriented verification based on record subtyping in higher-order logic. In: Grundy J, Newey M (eds) TPHOLs'98, 11th international conference on theorem proving in higher-order logics, Canberra, Australia, September 1998. Lecture notes in computer science, vol 1479. Springer, Heidelberg, pp 349–366

[Obu05] Obua S (2005) Proving bounds for real linear programs in Isabelle/HOL. In: Hurd J, Melham T (eds) TPHOLs'05, 18th international conference on theorem proving in higher-order logics 2007, Oxford, UK, August 2005. Lecture notes in computer science, vol 3603. Springer, Heidelberg, pp 227–244

[Obu07] Obua S (2007) Proof pearl: looping around the orbit. In: Schneider K, Brandt J (eds) TPHOLs'07, 20th international conference on theorem proving in higher-order logics 2007, Kaiserslautern, Germany, September 2007. Lecture notes in computer science, vol. 4732. Springer, Heidelberg, pp 223–231

[Obu08] Obua S (2008) Flyspeck II: the basic linear programs. PhD thesis, Technishe Universität München

[OS08] Owens S, Slind K (2008) Adapting functional programs to higher order logic. Higher Order Symb Comput 21(4):377–409

[RS97] Rubio J, Sergeraert F (1997) Constructive algebraic topology. Lecture notes summer school in fundamental algebraic topology, Institut Fourier, http://www-fourier.ujf-grenoble.fr/~sergerar/Summer-School/

[WB89] Wadler P, Blott S (1989) How to make ad-hoc polymorphism less ad-hoc. In: Conference record of the 16th annual ACM symposium on principles of programming languages. ACM, New York, pp 60–76

Received 20 March 2008

Accepted in revised form 16 June 2009 by M. Broy

Published online 15 July 2009