

Extracting Computer Algebra Programs from Statements^{*}

Jesús Aransay¹, Clemens Ballarin², and Julio Rubio¹

¹ Dpto. de Matemáticas y Computación, Univ. de La Rioja, 26004 Logroño, Spain
{jesus-maria.aransay, julio.rubio}@dmc.unirioja.es

² Institut für Informatik, Technische Univ. München, D-85748 Garching, Germany
ballarin@in.tum.de

Abstract. In this paper, an approach to synthesize correct programs from specifications is presented. The idea is to extract code from definitions appearing in statements which have been mechanically proved with the help of a proof assistant. This approach has been found when proving the correctness of certain Computer Algebra programs (for Algebraic Topology) by using the Isabelle proof assistant. To ease the understanding of our techniques, they are illustrated by means of examples in elementary arithmetic.

1 Introduction

Kenzo is a Common Lisp program created by Sergeraert [10], for Computer Algebra computations in the field of Algebraic Topology. Its main characteristics are its handling of infinite spaces (by using functional programming), and that Kenzo has found results unreachable by any other means (see [17]).

Taking into account these features, a project to analyze formally fragments of Kenzo was undertaken, with the aim of increasing the reliability of the system. Two kinds of formal methods have been applied. First, algebraic specification techniques have been used to model the data structures in algorithmic Algebraic Topology (see [13]). Second, we are using proof assistants to mechanize the reasoning needed in this area of Computer Algebra (see [1], [2], [3]).

Several approaches have been introduced to deal with objects (as chain complexes, morphisms, and so on) of Algebraic Topology in the Isabelle proof assistant [15]. One of the fundamental theorems in algorithmic Homological Algebra, namely the so-called *Basic Perturbation Lemma* [6], was chosen as a test case for these experiments. The final formalization considers morphisms as records encoding the real map, the potential source and target chain complexes, an also the real domain of definition and the image (see details in [3]).

Despite of the success of this approach, the price to be paid was that we got a formalization (and proof mechanization) of the *theorems*, but not of the *programs* appearing in Kenzo. To bridge the gap between (mechanized) theorems and programs, we considered to use Berghofer's tool [4] for extracting ML programs

^{*} Partially supported by SEUI-MEC, project TIC2002-01626.

from Isabelle theories. We suspected that the proving efforts previously done could be perhaps unsuitable, due to additional constraints on the constructive nature of the proofs (up to now, we have chosen a *classical* way of proving in Isabelle, trying to emulate the proofs-by-hand from Homological Algebra). Surprisingly enough, we observed that most of our already formalized theorems had *constructive statements* (in a sense which will be explained in this paper), even if proofs are not necessarily expressed in a constructive manner. This simple observation allows to apply Berghofer’s tool to some of our Isabelle theories, extracting ML programs equivalent to some (small) fragments of Kenzo. Even if preliminary (the programs extracted so forth are extremely simple, compared with Kenzo as a whole), these results invite to explore further this research line. This paper is devoted to illustrate, through simple examples, this approach.

The paper is organized as follows. Sections 2 and 3 briefly introduce, respectively, some mathematical definitions and the case study chosen from Kenzo: the composition of two morphisms. In Section 4, we move to a well-known domain, namely elementary arithmetic, to work out a simple example related to Euclid’s proof on the existence of infinitely many primes. The aim of this section is to introduce some key ideas, without the complexities of Homological Algebra and Algebraic Topology. Basics on formalization, automated theorem proving and program extraction are commented on in Section 5, where our notion of *constructive statement* is (informally) introduced, too. Then, this notion is applied in Section 6 to the elementary arithmetic example, showing how Berghofer’s tool can be used to obtain an ML program (certified correct) computing a prime number bigger than its input. In Section 7, we go back to Computer Algebra, applying the same techniques to obtain an ML program (certified correct with Isabelle) to compose two morphisms. The paper ends with a section devoted to conclusions and open problems.

2 Mathematical Preliminaries

The mathematical machinery necessary to deal with the objects in the Kenzo system is quite complicated. In order to make easier the reading of this short paper, we focus only on the composition of morphisms, used as an illustrating example. The essential definitions are the following.

Definition 1. A graded group C_* is a family of abelian groups indexed by the integer numbers, $C_* = \{C_n\}_{n \in \mathbb{Z}}$, with each C_n an abelian group. A graded group morphism $f: A_* \rightarrow B_*$ of degree k ($\in \mathbb{Z}$) between two graded groups A_* and B_* is a family of group morphisms, $f = \{f_n\}_{n \in \mathbb{Z}}$, with $f_n: A_n \rightarrow B_{n+k}$ a group morphism $\forall n \in \mathbb{Z}$. A chain complex is a pair (C_*, d_{C_*}) , where C_* is a graded group, and d_{C_*} (the differential map) is a graded group morphism $d_{C_*}: C_* \rightarrow C_*$ of degree -1 such that $d_{C_*}d_{C_*} = 0$. A chain complex morphism $f: (A_*, d_{A_*}) \rightarrow (B_*, d_{B_*})$ between two chain complexes (A_*, d_{A_*}) and (B_*, d_{B_*}) is a graded group morphism $f: A_* \rightarrow B_*$ (degree 0) such that $f d_{A_*} = d_{B_*} f$.

The mathematical result whose proof will be considered in the following sections is elementary: the composition of two morphisms is again a morphism.

Even if elementary, it is a fundamental fact which is used intensively in more interesting applications. In particular, it is instrumental in the proof (and even in the statement and in the algorithm) of the Basic Perturbation Lemma (BPL, from now on), central theorem [6] which has been studied in previous works, from several points of view (see [17], [3]).

3 The Kenzo Program

In [16] a fragment of the implementation in Kenzo of the BPL is presented. This is a quite complex Common Lisp program. As explained above, we focus here on the simpler case of the composition of morphisms. Even in this case, some explanations are needed in order to understand the code.

In Kenzo every chain complex is *free*. That implies a graded group in Kenzo is defined simply by a set of *generators* and an *equality test* among them, in *each degree*. The generators are used to form *combinations* (that are linear combinations of generators, with coefficients over the integer numbers), which are the real elements of the group in each degree. To add two combinations, the equality test between generators is used. With this organization, to define a morphism between chain complexes, it is enough to give the image of each generator, that will be a combination in the target group. In order to extend this map to combinations on the source group, the equality test *in the target group* is used. This is the most frequent *strategy* to define a morphism in Kenzo. But there are situations where this way of working is really wasteful from the performance point of view: think of the identity or the null morphisms, where obviously no equality checking is needed. So, Sergeraert considered a second strategy, called *by combination* (:cmbn as keyword in Common Lisp), to cover this case (the first, more frequent, strategy is called *by generator*, and denoted in Common Lisp by :gnrt). This explains the optional argument (called *strt* for *strategy*) in the following Kenzo program.

```
(DEFMETHOD CMPS ((mrph1 morphism) (mrph2 morphism) &optional strt)
  ;; ... lines skipped
  (build-mrph :sorc sorc2 :trgt trgt1 :degr (+ degr1 degr2)
    :intr #'(lambda (cmbn)
              (declare (type cmbn cmbn))
              (the cmbn
                 (cmbn-? mrph1 (cmbn-? mrph2 cmbn))))
    :strt :cmbn :orgn '(2mrph-cmps ,mrph1 ,mrph2 ,strt))
  ;; ... lines skipped
```

The program is a *method*, since Kenzo is built on CLOS (the Common Lisp Object System). This allowed Sergeraert, by using the inheritance mechanism of object-oriented programming, to give the same name to similar operations which compose two morphisms of coalgebras and another related algebraic structures. The lines skipped correspond to the cases in which one of the two morphisms has (or both have) a strategy by generator. The fragment showed here corresponds

to the case in which at least one of the morphisms has strategy by combination, and this is also the strategy in the result morphism: `:str` `:cmbn`. The symbols `src2`, `trgt1`, `degr1` and `degr2`, correspond, respectively, to the source of the second morphism `mrph2`, to the target of the first one, and to the degrees of the morphisms, which have been previously extracted from the morphisms in the lines skipped. The fragments `(declare (type cmbn cmbn))` and `the cmbn`, show that we are in a *typed* context. The keyword `:orgn` is used to store some information on the *origin* of the new morphism (that is to say, on the method and arguments constructing the new object), for software engineering purposes. Finally, the Kenzo function `cmbn-?` allows the programmer to invoke a morphism on a combination, and it is used here (twice) to accomplish the actual composition.

Apart from technicalities, the essence of the previous method, in the particular case of chain complexes morphisms of degree zero, is equivalent to the following Common Lisp function.

```
(defun CMPS (g f)
  (build-mrph :src (src f) :trgt (trgt g)
             :intr #'(lambda (cmbn)
                       (cmbn-? g (cmbn-? f cmbn)))))
```

The formalization work will be illustrated with this simplified version. But, instead of start with this algebraic case, we deal in the next section with an elementary example in number theory.

4 An Elementary Example

In this section, the same question of proving the correctness of a program is studied, but in the domain of elementary arithmetic, with the aim that our ideas can be understood without the complexities of Homological Algebra.

The example is related to prime numbers. The following Common Lisp program computes the next prime to a given positive integer number $x > 1$.

```
(defun nextprime (x) ; x integer > 1
  (if (isprime (+ x 1))
      (+ x 1)
      (nextprime (+ x 1))))
```

Here, we are assuming the existence of a Common Lisp program `isprime` used to determine if its input is a prime number. In addition, we assume that `isprime` is correct. This will be used in the following elementary result.

Theorem 1. *The program `nextprime` is correct with respect to the following specification:*

Input specification: x integer, $x > 1$.

Output specification: `(isprime (nextprime x))` returns true.

Proof. In two parts:

1) *Partial correctness.* If the program stops, $(\text{nextprime } x) = z$, with $z = (+ y 1)$ and $(\text{isprime } (+ y 1)) = \text{true}$ (by the semantics of `if`). Then, the output specification follows.

2) *Proof of termination.* By contradiction.

Let us assume that there exists x integer, $x > 1$, such that $(\text{nextprime } x)$ does not stop. This implies that $\forall y > x$, y is not a prime number. Let $P = \{p_1, \dots, p_n\}$ be the set of primes smaller than x . By hypothesis, this is the set of *all* prime numbers. Let us consider $m = p_1 * \dots * p_n + 1$. Then we have that p_i does not divide m , $\forall i = 1, \dots, n$. We conclude, by applying Lemma 1 (see below), that m is prime. But $m > p_i$, $\forall i = 1, \dots, n$, and thus $m \notin P$ and m is not a prime number. *Contradiction.* \square

Lemma 1. *Let m be an integer number, $m > 1$. Then, m is prime if and only if for all prime number $p < m$, p does not divide m .*

Everyone will perceive in this proof Euclid's argument to show that there are infinitely many primes. Nevertheless, there are several variations on Euclid's idea. This one has the property that the computational content is more hidden than in other variants, which are based on the following Lemma 2, instead of on Lemma 1. In those proofs, even if presented in a "reductio ad absurdum" manner, the computational content is quite explicit¹. More on that in Section 6.

Lemma 2. *For all integers $m > 1$, there exists a prime number $p \leq m$ such that p divides m .*

We have tried to write down a very detailed proof of correctness, since our aim is not only to give such a proof, but also a *mechanized certificate of correctness*. To be precise, we are looking for Isabelle [15] scripts containing proofs of correctness for our programs. To this aim, it is necessary to link in some way the running code (or, at least, the source code) with some formalization of it. In general, this is a task far from trivial (the lack of the sought link has been illustrated graphically in the previous proof, where x and x have been used in an indistinguished manner). Different approaches to formalize and mechanize such proofs are explored in the next section.

5 Formalization and Mechanization

In order to build correctness certificates for programs, a first necessary task is to formalize the objects of study in a computer-aided mathematical tool. There are many such tools: for instance, ACL2 [11], Coq [7], Mizar [18] or Isabelle [15]. The choice of one of them depends on several criteria, including the expressiveness

¹ Thanks are due to W. Bosma, who explained in the Map e-list (see <http://www.disi.unige.it/map/>) that the same is not true in the proof presented above.

of the underlying logic, the degree of automation or the libraries previously developed for our domain of interest. In the elementary example of the section above both Mizar and Isabelle could be convenient (in fact, the example and proof have been extracted from [21], where they are used to compare Mizar and Isabelle). However, once the objects of study have been formalized, more efforts are needed in order to link the formalization with a *real* program.

One well-known strategy is to establish the formalization in a (mechanized) *constructive* logic, and then extract a program from the proof. This is the point of view when using the Coq system [7] for this task. Nevertheless, it is not the *only* way to get certificates for programs.

In particular, the Common Lisp program `nextprime` in Section 4, with its non-constructive proof, seems to be far from this kind of approach. This is true from a strict-constructivist perspective, but it is not in Markov's *constructive recursive mathematics*². In fact, the program `nextprime` is a paradigmatic example of a (correct) program generated by *Markov's principle* [14]. These issues are also discussed in [12], where Markov's principle is integrated with constructive type theory, and by Berghofer in [5], in the context of his tool to extract code from Isabelle scripts.

Taking into account this example it seems that Davenport's observation in [8] (or, even more explicitly, in page 140 of [9]), claiming that *Computer Algebra* correctness proofs can be done by not necessarily *strict-constructive* methods, is quite accurate (up to our knowledge, if proofs in Computer Algebra can go beyond Markov's constructivism is an open problem).

In fact, in the field of algorithmic homological algebra it has been observed that the theorems to be formalized have *constructive statements*. That is to say: a new object is defined (the composite of two morphisms, in our running example) and some property of this object is asserted (namely, it is a morphism). Then, code can be extracted from the definition or specification appearing in the statement. Thus, this approach seems to indicate that the underlying logic in the proof is not mandatory to be constructive (since the program is extracted from definitions and not from proofs), in the vein of Davenport's observation. This strategy will be presented in the case of the composition in Section 7. But, first, we go back to the arithmetical example, to show how in this case statements can be rendered *constructive*, allowing program extraction in presence of classical proofs.

6 The Elementary Example Revisited

In this section the example in Section 4 is reconsidered. The idea is that the computational content of a proof (presented in a constructive or in a non-constructive manner) can be made explicit in order to build a new statement, which is *constructive* in the informal sense introduced above.

² We are grateful to F. Sergeraert who attracted our attention to this important variant of constructivism.

We use Lemma 2 in Section 4 above to define a primitive recursive function “some-prime-divisor” that for each integer number greater than 1 computes a prime divisor of it.

primrec

```
some-prime-divisor-aux x 0 = 0
some-prime-divisor-aux x (Suc n) =
  (if (prime-cons (Suc n) ∧ dvd-cons (Suc n) x) then (Suc n)
    else some-prime-divisor-aux x n)
```

consts

```
some-prime-divisor :: nat => nat
```

defs

```
some-prime-divisor-def: some-prime-divisor x == some-prime-divisor-aux x x
```

consts

```
some-big-prime :: nat => nat
```

defs

```
some-big-prime-def: some-big-prime x == some-prime-divisor (x! + 1)
```

The same task of *rebuilding* must be done with the predicates “divides” and “isprime”, which are converted to two (Boolean-valued) functions “dvd-cons” and “prime-cons” (here *cons* stands for *constructive*). In addition, it is necessary to prove in Isabelle that these functions have the right properties (by relating them, for instance, to the predicates “dvd” and “prime” from the Isabelle libraries on prime numbers). With these preparations, the following lemma can be proved in Isabelle.

theorem $x < (\text{some-big-prime } x)$

proof (*unfold some-big-prime-def*)

let $?p = \text{some-prime-divisor } (x! + 1)$

from *some-prime-divisor-gt-zero*

have *prime-cons*: $\text{prime-cons } ?p$ **and** *dvd-cons*: $\text{dvd-cons } ?p (x! + 1)$

by (*simp-all add: some-prime-divisor-properties*)

from *prime-cons* **have** *prime-p*: $?p \in \text{prime}$

by (*simp add: prime-equiv-prime-cons*)

from *dvd-cons* **have** *dvd*: $?p \text{ dvd } (x! + 1)$

by (*simp add: dvd-cons-impl-dvd*)

show $x < ?p$

proof –

have $\neg ?p \leq x$

proof

assume $?p \leq x$

with *prime-g-zero* **and** *prime-p* **have** $?p \text{ dvd } x!$

by (*simp add: dvd-factorial*)

with *dvd* **have** $?p \text{ dvd } (x! + 1) - x!$ **by** (*rule dvd-diff*)

then have $?p \text{ dvd } 1$ **by** *simp*

with *prime-p* **show** *False* **using** *prime-nd-one* **by** *auto*

qed

then show *?thesis* **by** *simp*

qed

qed

This proof, a variant of Euclid’s argument, is an adaption of a proof script presented in [21]. It obviously has a non-constructive presentation. It might be considered unsuitable from a constructivist’s point of view, but it is well-known and easily understood. The important observation now is that, no matter the logic underlying the proof, the *statement* is constructive, and Berghofer’s tool [4] can be applied on the complete theory (only the final fragment is displayed here) to obtain the following ML program (the names for certain constants have been modified, in order to make the code more readable):

```

fun some_prime_divisor_aux x 0 = 0
  | some_prime_divisor_aux x (Suc n) =
    (if (prime_cons (Suc n) andalso dvd_cons (Suc n) x) then Suc n
      else some_prime_divisor_aux x n);

fun some_prime_divisor x = some_prime_divisor_aux x x;

fun fact 0 = 1 | fact (Suc n) = times (fact n) (Suc n);

fun some_big_prime x = some_prime_divisor (plus (fact x) 1);

```

This ML code is to be compared to the Common Lisp program `nextprime` in Section 4. Both functions compute a prime number greater than its argument x . But of course, `nextprime` is quite more efficient.

7 Application to Computer Algebra

When applying these ideas to the Kenzo program, the first observation to be made is that the elaboration done in the previous section is unnecessary: many of the theorems to be proved have already a *constructive statement* (or can be easily transformed into such a statement). See details in [3]. Therefore, the same work already made for the formalization, can be reused for extracting code from statements.

As explained at the end of Section 3, we will work with a simplified version of the Kenzo composition. More concretely, we do not consider the degree in the structures (that is, the degree of morphisms is 0) and, in addition, we do not assume that groups are free (that is to say, the strategy is always `:cmbn`, by *combination*, following Kenzo terminology). One fragment of such a formalization can be found here:

```

constdefs
  group-mrp-comp :: [ ('b, 'c) group-mrp-type, ('a, 'b) group-mrp-type ] =>
    ('a, 'c) group-mrp -type
  group-mrp-comp g f ==
  (| src = src f, trg = trg g, morph = (morph g) o (morph f),
    src-comm-gr = src-comm-gr f, trg-comm-gr = trg-comm-gr g |)

```

lemma *group-mrp-composition*:
assumes *A1*: *group-mrp A*
and *B1*: *group-mrp B*
and *C1*: *trg-comm-gr A = src-comm-gr B*
and *D1*: *trg A = src B*
shows *group-mrp (B o A)*

We can now apply Berghofer’s extraction tool [4] on the definition appearing in the statement, obtaining the following ML program (only the most relevant part is shown here):

```
fun comp g f = (fn x => g (f x));

fun group_mrp_comp g f =
  group_mrp_type_ext (src f) (trg g) (comp (morph g) (morph f))
    (src_comm_gr f) (trg_comm_gr g) Unity;
```

In this case, the proof of the previous Isabelle lemma is to be considered as a proof of correctness for the ML program (assuming, as usual, the soundness of Berghofer’s translation). This (certified correct) ML program should be compared with the *real* corresponding Kenzo program or better, to ease the reading, with the simplified Common Lisp version given at the end of Section 3.

8 Conclusions and Future Work

In this paper we have applied Berghofer’s extraction tool for Isabelle scripts to obtain Computer Algebra programs, with a certificate of correctness. Our contribution lies on extracting code from *statements* and not from *proofs*, as usual in constructive type theory. From a technical point of view, it would be more accurate to say that code is extracted from *definitions* appearing in statements, but it has been considered more appealing to exploit the couple statement/proof. In fact, in the arithmetic example in Section 6, it is clear that we are *programming* inside Isabelle, by transforming predicates into recursive functions, and proving, at the same time, the correctness of these transformations.

Even if it seems that in the elementary examples from Computer Algebra in Algebraic Topology, this work of *programming in Isabelle* is not necessary, important challenges are still open. It would be necessary to bridge the gap between the ML and Common Lisp programming language, and even more difficult, the gap between the ML programs extracted (that could be very inefficient) and the corresponding performing programs which are really usable. With respect to this, the toy example with prime numbers could illustrate the strong difficulties (in terms of proving efforts within Isabelle) to obtain a reasonably efficient program. Thus, finally it will be perhaps unavoidable to *program in Isabelle* in order to get usable programs. From our point of view, this problem of using proof assistants to synthesize “real-life” programs is a central one in intelligent information processing (see, for instance, [19] and [20]).

References

1. J. Aransay, C. Ballarin and J. Rubio, *Deduction and Computation in Algebraic Topology*, Proceedings IDEIA 2002, Universidad de Sevilla (2002) 47-54.
2. J. Aransay, C. Ballarin and J. Rubio, *Towards a higher reasoning level in formalized Homological Algebra*, Proceedings Calculemus 2003, Aracné Éditrice (2003) 84-88.
3. J. Aransay, C. Ballarin and J. Rubio, *Four approaches to automated reasoning with differential algebraic structures*, Lecture Notes in Artificial Intelligence 3249 (2004) 222-235.
4. S. Berghofer, *Program Extraction in Simply-Typed Higher Order Logic*, Lecture Notes in Computer Science 2646 (2002) 21-38.
5. S. Berghofer, *Answer to Tom Ridge*, `isabelle-users@cl.cam.ac.uk`, February 18, 2005.
6. R. Brown, *The twisted Eilenberg-Zilber theorem*, *Celebrazioni Arch. Secolo XX, Simp. Top.* (1967) 34-37.
7. *The Coq Proof Assistant Reference Manual*, <http://coq.inria.fr/doc/main.html>
8. J. M. Davenport, *Effective Mathematics: the Computer Algebra viewpoint*, Lecture Notes in Mathematics 873 (1981) 31-43.
9. J. M. Davenport, *Algebraic computations and structures*, Lecture Notes in Pure and Applied Mathematics 113, Marcel Dekker (1989) 129-144.
10. X. Dousson, F. Sergeraert and Y. Siret, *The Kenzo program*, <http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/>
11. M. Kaufmann, P. Manolios and J. Strother Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.
12. A. Kopylov and A. Nogin, *Markov's principle for propositional type theory*, Lecture Notes in Computer Science 2142 (2001) 570-584.
13. L. Lambán, V. Pascual and J. Rubio, *An object-oriented interpretation of the EAT system*, *Applicable Algebra in Engineering, Communication and Computing* 14 (2003) 187-215.
14. A. A. Markov, *On constructive mathematics*, AMS Translations 98 (1971) 1-9.
15. T. Nipkow, L. C. Paulson and M. Wenzel, *Isabelle/HOL: A proof assistant for higher order logic*, Lecture Notes in Computer Science 2283, 2002.
16. J. Rubio, *Constructive proofs or constructive statements?*, Dagstuhl Proceedings 05021, 2005. <http://www.dagstuhl.de/05021/>
17. J. Rubio and F. Sergeraert, *Constructive Algebraic Topology*, *Bulletin des Sciences Mathématiques* 126 (2002) 389-412.
18. P. Rudnicki, *An Overview of the MIZAR Project*, In Proceedings Workshop on Types for Proofs and Programs (1992) 311-330. <http://web.cs.ualberta.ca/piotr/Mizar/MizarOverview.ps>
19. N. Schirmer, *A verification environment for sequential imperative programs in Isabelle/HOL*, Lecture Notes in Computer Science 3452 (2005) 398-414.
20. M. Takeyama, Q. Haiyan, P. Dybjer, *Verifying Haskell programs by combining testing, model checking and interactive theorem proving*, *Information and software technology* 15 (2004) 1011-1025.
21. M. Wenzel and F. Wiedijk, *A Comparison of Mizar and Isar*, *Journal of Automated Reasoning* 29 (2002) 389-411.