A type for Taylor series for the `C++` library `iRRAM` for exact real arithmetic

Florian Steinberg

last edited February 14, 2013

Contents

I Higher type computability theory **3**

1 Computable analytic functions **3**

1.1 The constants k and A 3

II `iRRAM` **4**

2 Basic data types **6**

2.1 The class REAL 6

2.2 The class COMPLEX 6

2.3 The class INTEGER 6

3 `iRRAM` functions **6**

3.1 Output functions 6

3.2 `iRRAM::FUNCTION`s 7

3.3 Limits 7

4 Tools from the `C++11` standard template library **7**

4.1 `std::functions` 8

4.2 The `C++` lambda calculus 9

4.3 `std::shared_ptrs` 10

III Implementation **11**

5 Some classes of functions and similar objects **11**

5.1 The class `POLY` 11

5.2 The class `FUNC` 13

5.3 The class `POWERSERIES` 15

5.3.1 The class `coeff_fetcher` 16

6 A class for Taylor series **17**

6.1 The class definition 17

6.2 The evaluation 17

Introduction

We aim to implement a type of analytic functions in `iRRAM`. The `iRRAM` is a `C++` package for error-free real arithmetic. We will begin by summarizing the facts about computable analytic functions we need. We will go on to describe some of the key features of `iRRAM`. The we will present some parts of the `C++11` standard template library we need for implementation. Finally we will describe how the implementation was done. In the last chapter we will address some shortcomings and possible future improvements.

Part I
Higher type computability theory

1 Computable analytic functions

We will only consider analytic functions, that are equal to their Taylor expansion around zero on a open superset of the Unit disc. This means, that we can find some $r > 1$ which is still strictly smaller than the radius of convergence. In the following we will fix such a function f.

f is uniquely determined by its Taylor-series in zero. In the following, we will denote the Taylor-series of f by $(a_n)_{n\in\mathbb{N}}$. It can be expressed in terms of fs derivatives, or by Cauchy's differentiation formula:

$$a_n = \frac{f^{(n)}(0)}{n!} = \frac{1}{2\pi i}\int_{|z|=r} \frac{f(z)}{|z|^{n+1}} d\lambda.$$

The sequence $(a_n)_{n\in\mathbb{N}}$ is computable (as sequence of real numbers) if and only if f is computable (as continuous function).

1.1 The constants k and A

Unfortunately it is not possible to evaluate such a function effectively without further information. We additionally need two constants k and A. k will be such, that $r := \sqrt[k]{2}$ is still smaller than the radius of convergence of $(a_n)_{n\in\mathbb{N}}$ and A such that for all $n \in \mathbb{N}$

$$|a_n|r^n \leq A.$$

Since we demanded a convergence radius strictly larger than 1, constants like this will always exist.

We will briefly discuss how these constants can be found: Since the radius of convergence of f is assumed to be strictly larger than 1, and $r := \sqrt[k]{2}$ goes to 1 as k goes to infinity, it is possible to choose k big enough for r to be in between 1 and the radius of convergence. Now fix such an k. Consider the function

$$f|_{\{z:|z|=r\}}.$$

Since this is a continuous function on a compact domain, it will be bounded. If A is a bound of this function, the Cauchy differentiation formula gives:

$$|a_n| = \left| \frac{1}{2\pi i} \int_{|z|=r} \frac{f(z)}{|z|^{n+1}} d\lambda \right| \leq \frac{A}{r^n}$$

Thus A is as desired.

Using these constants, we can obtain a tail estimate:

$$\left| \sum_{n \geq N} a_n z^n \right| \leq A \frac{(|z|/r)^N}{1 - |z|/r}.$$

In particular we get a bound for f on the unit disc: If $|z| \leq 1$, we get:

$$|f(z)| \leq A \frac{r}{r-1}.$$

More information on how to calculate these constants (for example for the product of two such functions) can be found in **??**. The most important part of the source for our intentions will be Theorem 3.3. The proof of this Theorem in particular specifies how constants k' and A' for the derivative f' can be found, namely:

$$k' := 2k$$

and

$$A' := \left\lceil \frac{A}{r} \left(1 + \frac{2k}{e \ln(2)} \right) \right\rceil.$$

Since any bound of f' on the unit disc is also a bound of the Lipschitz constant of f, this allows us to explicitly calculate a Lipschitz constant

$$L := \left\lceil A \frac{\left(1 + \frac{2k}{e\ln(2)}\right)}{r - \sqrt{r}} \right\rceil$$

for f with very little computational effort.

We have used the ceiling function above. In terms of computable analysis, this function is not well behaved (since it is not continuous, it will not be computable). Thus we will use the function $x \mapsto$ `round2`$(x)+1$ as a replacement. Here `round2`() is a multivalued function already implemented in `iRRAM`.

Part II

`iRRAM`

At first glance the most reasonable approach to computable analysis would be to represent a real number x by an algorithm P taking a natural number n and returning an approximation, i.e. a appropriately encoded rational number x_n such that $|x - x_n| < 2^{-n}$. This kind of proceeding bears the following problems:

- Each time a sum or a product of real numbers is calculated, the algorithms of the corresponding real numbers must be copied or at least referenced. This leads to a tree-like structure of any program and often to uncontrollable growth of memory consumption.

- A algorithm P encoding a real number carries more information than the real number itself. If P is given to a function, this function can for example also use the running time of P to generate its output. This kind of functions should from a mathematical point of view not be considered computable, and can lead to pathological behavior.

Thus the `iRRAM` chooses a different approach. Namely the real numbers are represented by finite intervals containing said real number. All manipulations are carried out with these intervals. If a situation occurs, where the precision is simply not sufficient anymore, the whole calculation is restarted with higher precision. This procedure is called a *reiteration.* The single runs of the program with different precisions will be referred to as *iterations.*

At first glance, this approach seems to be very time consuming: The whole computation might be restarted repeatedly, discarding all the computations made in the earlier iterations completely. But it is well known, that this does not blow up the asymptotic complexity: More precisely, the asymptotic complexity of the whole computation and the last reiteration coincide [].

`iRRAM`s approach does not suffer the problems listed above, but brings its own (which are hopefully more manageable):

- If the program includes in and output, a restart of the program will lead to doubled output. This is mainly a implementation problem and can be avoided by using the output methods provided by `iRRAM`.

- If on the other hand, the program asked the user for input at some point, this input will have to be memorized. Moreover: if any multivalued function is computed, it has to evaluate to the exact same value in the next run and has to be memorized. This is to avoid incoherences in output.

- Each time a reiteration is triggered, the whole program restarts. This means, that nearly everything (as mentioned above some things are memorized) is reevaluated. In particular: If the program is composed of two tasks, and one of those needs a higher precision, both tasks will be carried out in this higher precision. This means, that reiterations are very expensive in terms of computation time.

Another point is, that programs for `iRRAM` are written in `C++`. Since `C++` is a very powerful programing language, it is often possible for the user to do things he is not really supposed to do. We will encounter this problem ourselves in **??**.

We will now consider some parts of `iRRAM` more closely.

2 Basic data types

2.1 The class REAL

2.2 The class COMPLEX

2.3 The class INTEGER

3 `iRRAM` functions

3.1 Output functions

We did already mention, that output can be a problem and that one should use the output methods provided by `iRRAM`. To cope with the difficulties of duplicated output, `iRRAM` has its own class of output streams `iRRAM`::`orstream`, which replaces the standard output streams. The following can be used for output:

- Any `iRRAM`::`orstream` via the overloaded `<<` operators. In particular the standard one `iRRAM`::`cout`.
- The functions `iRRAM`::`rwrite`, `iRRAM`::`rwritee` and `iRRAM`::`rshow`.
- The function `iRRAM`::`rfprint`.

In the following we will leave the preceding `iRRAM`:: away. This coincides with the syntax used inside of `iRRAM` programs. One should note, that `cout` differs from `std::cout`: For example pointers will be output as 1 if they point somewhere and as 0 if they do not. This is important, since pointer addresses can change in reiterations, which could lead to inconsistent output. Additionally `cout` might change the path of computation, since it can trigger reiterations and there are situations where `cout` does not lead to output at all (for example in passages where reiterations might be triggered). Thus when changing `iRRAM` itself, it is sometimes a good idea to still use `std::cout` for debugging purposes.

The first point in the list above does play a special role here, since it is used by all the others. They call the function `swrite`, which prints a `REAL` to a string and then output this value through the `cout` by using the `<<` operator.

Thus we will first take a closer look at the class `orstream`. The main components of an `orstream` are:

`target`: An `std::ostream` pointer, used for output.

`requests`: A static but thread specific request counter.

`outputs`: A static but thread specific output counter.

`real_w`: A standard value for the output width of real numbers.

`real_f`: **??**.

The output operator `<<` is defined for most standard data types of `C++`, and for the `iRRAM` specific data types like `REAL`, `DYADIC`, `RATIONAL`, `INTEGER` and `COMPLEX`.

Lets assume the operator `<<` is called with some `orstream ors` and some standard `C++` data type `x`. `ors` will then increase the counter `requests`, check if `requests` is bigger than `outputs`, and if so to pass `x` to the `std::ostream` pointed at by target and also increase the counter `outputs`. If now a reiteration occurs, the counter `requests` will be reset, but the counter `outputs` will be kept. This leads to the following behaviour: In the next iteration first `outputs` outputs will be ignored, these are evidently exactly those which where already printed in one of the earlier iterations.

If the operator `<<` is called with some `x` of `iRRAM` specific data type, then `ors` will call the function `swrite` with parameters `x` and `real_w`, which will print `x` to a string `xs` of length `real_w`, but at least 9 characters. Then `ors` will call `ors << xs`. Since the precision of `x` may not suffice to extract a string representing `x` of the desired length, `swrite` might trigger a reiteration.

We emphasize once more: It is important to acknowledge the restrictions of the `orstreams`. If you could output pointer addresses, these would change in reiterations. Since output once written will not be revised, these might be wrong in later iterations. In this case the `iRRAM` takes care of the problem by not outputting pointer addresses but first casting them to `bools`. A real example: if you output the error of some REAL it might show values greater than one, which might not reflect the future behaviour as the very next command could be to output the REAL, which will then trigger a reiteration and increase precision. Since the output will not be revised the result could be confusing for the user. This is put down in red, since it might be considered a bug and removed in future versions. Although these problems should be handled by `iRRAM` itself, it is clear that there will always be some ways to trick the system and it is advised to handle output with care.

The remaining output functions simply call the function `swrite` with the desired precision and then hand the string to `iRRAM`s standard `orstream cout`.

3.2 `iRRAM::FUNCTION`s

We aim to implement a type for analytic functions in `iRRAM`, which is not yet present. But there already is a type of functions implemented. We will give a short review of this type.

3.3 Limits

4 Tools from the `C++11` standard template library

In `C++`, there are four kinds of functional objects: functions, member functions, function pointers and member function pointers. Since the first two are very common, we will start at function pointers. It is important to note, that function pointers, in contrast to regular pointers, do not point to a chunk of memory where a function is located, but to a the piece of code where the function is defined. This makes it impossible to dynamically create new functions when needed, which is a serious shortcoming for us, since we want to be able to add and multiply functions. Member function pointer actually solve this problem, since classes, and with them their member functions, can be dynamically created and destroyed. Really using member functions would be very involved, luckily for us `C++11` added improved syntax for exactly this.

We will in the following sections review the tools we need to handle functions and their dependencies.

4.1 `std::functions`

The std::function template is a general-purpose polymorphic function wrapper (according to cppreference.com). We will find `std::functions` to be highly useful. But it will be not until we learn about the `C++` lambda calculus, that we can grasp their whole potential. Thus the description in this chapter might appear somewhat unspectacular. A `std::function` can be defined by

```
std::function<RESULT(PARAM)> f;
```

A std::function can be evaluated like a function, and defined from a function pointer, as the following short example shows:

```
#include<functional>

using std::function;

double f(int i) {
  return double(i);
}

int main()
{
  function<double(int)> g = f;
  cout << g(4) << endl;
  return 0;
}
```

The `std::function` can do a lot more than regular functions though. For example: If `f` is a `std::function<RESULT(PARAM1, PARAM2)>` of two arguments, we can define a `std::function<RESULT(PARAM2)> g` by setting the first parameter to a fixed value (say `x`) using the function `std::bind`:

```
function<RESULT(PARAM2)> g = bind(f, x, std::placeholders::_1);
```

To achieve something similar with regular function pointers is complicated (though it is possible). We will see that some of those possibilities hold risks, since `iRRAM` might not expect to encounter a function, containing real numbers that are not handed to it as an parameter.

The syntax `function<RESULT(PARAM)>` of `std::functions` seems nicer than the `FUNCTION<PARAM, RESULT>` we have seen before. It is also more convenient, since confusing parameter and result type gets a lot harder. The former syntax is implemented by a template specialisation. The definition of looks like this:

```
template<class T>
class function {};

template<class T, class S>
class function<T(S)> {/*...*/};
```

Here the first two lines define an empty template class, then lines four and five specialize the definition in the case, that the template argument is of a specific form. Namely a string of the form T(S), where T and S are some arbitrary types. We will use this trick to also improve the syntax of our function type.

`std::functions` are in many respects superior to `C++` functions, but they lack one thing functions do have: the possibility to be made templates.

4.2 The `C++` lambda calculus

One of the main sources for `std::functions` is the `C++` lambda calculus. A lambda expression in `C++` is of the form

```
[/*captures*/](/*parameters*/) -> /*output_type*/ {/*algorithm*/};
```

Where the commented parts are to be replaced as follows:

`/*captures*/` is to be replaced by a list of variables of local scope, that are needed by the algorithm but supposed to appear as parameters of the function (those are actually what mathematicians mean, when they say 'parameters'). A variable occurring in this list will be called captured (by the lambda function). Global variables need not be captured and can be accessed from the algorithm anyway. Variables may be captured by copy or, if they are preceded by an '`&`', by reference.

`/*parameters*/` is to be replaced by the list of parameters of the function to be constructed.

`/*output_type*/` is to be replaced by the output type of the function to be constructed (if the value that follows the return command is not of this type, a implicit type conversion will be attempted). This part (together with the '`->`') can be omitted if the output type will be clear from the context.

`/*algorithm*/` is to be replaced by the algorithm calculating the return value from the parameters and the captured variables.

an easy example of a definition of a `std::function` via the `C++` lambda calculus could look like this:

```
int i = 5;
std::function f<double(int)> = [i](const int& n) {i*log(n);};
```

Here the output type was omitted, since it is specified in the `std::function`.

We remark, that members can not be captured directly: if `c` is an object of an class `C`, which has a member k of type T, then

```
[c.k]() -> T {return c.k;};
```

will not work. The reason is the following: Assume the class C has some member function f changing the value of k. In this case

```
[&c, c.k]() -> T {c.f(); return c.k;};
```

would be ambiguous, since c.k could mean both the field of the object c captured by reference, which has the new value, or the member c.k, which was captured by copy before the change was made and therefore should have the old value.

Thus we will have to first copy the member to a local variable, then capture it.

4.3 `std::shared_ptrs`

A shared pointer is an object consisting of an pointer (to an object of specified type) and a pointer to an reference counter. Each time the shared pointer is copied the reference counter will be increased. If a shared pointer is destroyed, it decreases the reference counter and checks if it is zero, and if it is, it destroys the object it is pointing to.

Shared pointers are very useful for treelike ownership relations: If one object is used by multiple other objects, each of the latter can, instead of an pointer or a copy, to the former hold a shared pointer. This way it is guaranteed, that we neither have useless copies, nor memory leaks because no one feels responsible for destroying.

The `std::shared_ptr` can be dereferenced by a preceding $*$, just as a regular pointer, can be constructed from regular pointers and compared to the `NULL` pointer. Here is an short example:

```
double* ptr = new double(4.5);
std::shared_ptr<double> s_ptr(ptr);
cout << *s_ptr << endl;
```

returns '4.5'.

There are two main sources of errors when handling shared pointers:

- Multiple shared pointers are constructed from the same pointer: In this case each shared pointer keeps its own reference counter. If the first of those counters hits zero, the object will be destroyed. If now a shared pointer following an other reference counter tries to access the object, there will be an error.

- Circular ownership relations: for simplicity lets have a look at the case of two lonely shared pointers pointing at each other. Each of the pointers has a reference counter which is one (since they are lonely). now assume we destroy one of them. This will decrease the corresponding reference counter and, since the reference count will hit zero, destroy the other shared pointer. This in turn will decrease its reference counter, which will also hit zero. Therefore it will attempt to destroy the first shared pointer. Which is already destroyed. This will lead to an error.

Part III
Implementation

5 Some classes of functions and similar objects

Before we talk about the class of Taylor series we are aiming to implement, we will introduce two more general classes (and one more restrictive). These classes are convenient for us, since a lot of code, which would otherwise make the class of Taylor series huge, can be 'exported'. This improves code readability. Of course we also hope that these classes might proof useful on their own.

Before we take a closer look at the individual classes, we give a short overview:

`POLY<coeff_type>` will be a minimalistic class for polynomials with coefficients of type `coeff_type`.

`FUNC<RESULT(PARAM)>` will be a new class for functions. Its functionalities will be very similar to those of `FUCNTION<PARAM, RESULT>`, but the implementation will be somewhat different: It will heavily be relying on the tools provided by the `C++11` standard template library.

`POWERSERIES<coeff_type>` will implement formal power series. This is, functions from the integers to objects of type `coeff_type` but with the convolution replacing the point wise product and the function `get_coeff` to avoid the operator `()`, which in this case might be ambiguous, since it could mean both the evaluation as function from the integers to `coeff_type` or analytic function. Also a formal derivative and anti derivative will be implemented.

There will be one additional helper class `coeff_fetcher<coeff_type>`, to improve the speed at which the coefficients of `POWERSERIES` can be evaluated.

To keep the class definitions presented in the following subsections readable, they might differ slightly from those, that can be found in the code. Some of the details left out will be addressed in later sections.

5.1 The class `POLY`

This class is supposed to model polynomials with coefficients from some class `coeff_type`. Its main component is a shared pointer named `coeff`, which points to a constant `std::vector<coeff_type>`. The vector is a constant, so will not have to copy it if we copy the polynomial. We will just copy the shared pointer. If we need to change the coefficients, we will create a new vector and reassign the shared pointer. If the polynomial was the only one referencing the former vector, this vector will be automatically deleted. The class definition of `POLY` looks similar to this:

```
 1 template<class coeff_type>
 2 class POLY {
 3   // members:
 4     private:
 5       shared_ptr<const vector<coeff_type>> coeff;
 6   // constructors:
 7     public:
 8       POLY();
 9       template<class T>
10       POLY(const T&);
11       POLY(vector<coeff_type>);
12   // standard operators:
13     public:
14       POLY& operator = (const POLY<coeff_type>&);
15       POLY operator - (const POLY<coeff_type>&);
16       /* ... */
17       POLY& operator *= (const POLY<coeff_type>&);
18   // member functions:
19     public:
20       template<class ARG>
21       ARG operator () (const ARG&);
22       unsigned int get_degree() const;
23       coeff_type get_coeff(const unsigned int n) const;
24 };
```

We briefly discuss the general purpose of the components of this class, before we address some of the key implementations.

`members:` The one existent member has already been discussed above.

`constructors:` In line 8 the empty constructor can be found, it will construct the zero polynomial. Then there is a constructor template in line 9 and 10, which will attempt to convert any given type `T` to `coeff_type` and then construct the polynomial having this value as first and only coefficient. Finally in line 11 we find a constructor, that constructs a polynomial from a given list of coefficients. We refrain from handing the vector over as a reference to allow syntax such as `POLY<REAL> P({1,2,pi()})`, where `{1,2,pi()}` as a temporary object can not be referenced.

`standard operators:` There is no complete list of the operators above, so we give one here: The operators `=` (copy constructor), `-` (additive inverse), `+`, `-` (substraction), `*`, `+=` and `*=` are overloaded for polynomials.

`member functions:` line 20 and 21 are taken by an evaluation operator. Since polynomials are often evaluated in types more general than `coeff_type` (for example Polynomials with integer coefficients are regularly taken as functions on the real or complex numbers) this operator is a template.

Of course the argument ARG needs to provide an addition and multiplication. Additionally ARG needs to have an multiplication with coeff_type from the right (this is in particular the case if ARG is constructible from coeff_type).

The implementation of the constructors is straight forward. So is most of the implementations of the standard operators. Since we will use the convolution product multiple times, we take a closer look at the multiplication:

```
POLY operator * (const POLY<coeff_type>& P) {
  vector<coeff_type> coeff_new;
  unsigned int degree_new = get_degree() + P.get_degree();
  for (unsigned int k = 0; k <= degree_new; k++) {
    coeff_type kth_coeff(0);
    for (unsigned int l = max(0,(int)k-(int)P.get_degree()); (l <=
        get_degree())&&(k-l<=P.get_degree()); l++) {
      kth_coeff += get_coeff(l) * P.get_coeff(k-l);
    }
    coeff_new.push_back(kth_coeff);
  }
  return POLY<coeff_type>(coeff_new);
}
```

... maybe this is not such a good idea, the convolution is nicer

5.2 The class FUNC

This class is supposed to model (computable) functions which maps objects of some class PARAM to objects of some other class RESULT. It will mainly consist of an shared pointer algorithm to a std::function<RESULT(const PARAM&)>, we will refer to as the algorithm of the function. Since algorithm will be passed on, if a new function is created from existing ones, it will point to a constant. Changing the algorithm will be done by creating a new function and reassigning algorithm. If the algorithm is not used by any other function, it will automatically be removed. Here is the class definition of FUNC:

```
template<class PARAM, class RESULT>
class FUNC<RESULT(PARAM)> {
  // members:
    protected:
      shared_ptr<const function<RESULT(const PARAM&)> algorithm;
  // constructors:
    public:
      FUNC(const alg_func<PARAM, RESULT>& f);
    protected:
      FUNC();
  // standard operators:
    public:
      FUNC& operator = (const FUNC<RESULT(PARAM)>&);
      FUNC operator = (const function<RESULT(PARAM)>&);
      friend FUNC<RESULT(PARAM)> operator - <> (const FUNC<RESULT(
          PARAM)>&);
      /*...*/
      friend FUNC<RESULT(PARAM)> operator - <> (const FUNC<RESULT(
          PARAM)>&,const FUNC<RESULT(PARAM)>&);
18        template<class PAR>
19        FUNC<RESULT(PAR)> operator () (const FUNC<PARAM(PAR)>&);
20    // member functions:
21      public:
22        RESULT operator () (const PARAM&) const;
23  };
```

We will first discuss the general purpose of the components of this class, and then address some of the key implementations.

`members`: In line 5 the main component of the class can be found: The shard pointer to the algorithm of the function.

`constructors`: There are two main constructors: The one listed in line 8 will construct a `FUNC` from an algorithm. There is an empty constructor in line 10. This constructor is protected, since a FUNC without an algorithm is pretty much useless. But it might be convenient to have an empty constructor accessible for the derived classes.

`standard operators`: All the standard operators will be overloaded for `FUNC`s. For improved symmetry the operators are external and thus represented by friend templates in the class definition. The composition is a exception to this rule: Since it is not possible to define new operators, the composition (line 22) will have the syntax $f(g)$ instead of $f \circ g$. This requires it to be a member of `FUNC`, since the operator `()` has to be.

`member functions`: The only member function is the evaluation, which should be self-explanatory.

The implementations of this class are very straight forward. We will only take a look at the composition, as an example of how to combine shared pointers to `std::function`s and the `C++` lambda calculus:

```
1  template<class PARAM, class RESULT>
2  template<class PAR>
3  FUNC<RESULT(PAR)> FUNC<RESULT(PARAM)>::operator () (
4    const FUNC<PARAM(PAR)>& f
5  ) {
6    if((algorithm == NULL)||(f.algorithm == NULL))
7      return FUNC<RESULT(PAR)>();
8    alg_ptr<PARAM, RESULT> _algorithm = algorithm;
9    alg_ptr<PAR, PARAM> f_algorithm = f.algorithm;
10   alg_ptr<PAR, RESULT> new_algorithm(new const auto function(
11     [_algorithm, f_algorithm](const PAR& x) -> RESULT {
12       return (*_algorithm)((*f_algorithm)(x));
13     }
14   ));
15   return FUNC<RESULT(PAR)>(new_algorithm);
16 }
```

Here we used the abbreviation `alg_ptr<PARAM, RESULT>` for `shared_ptr<const function<RESULT(const PARAM&)>>`. First we check, that both the functions have algorithms (line 6). If one of them does not, we return a function without an algorithm. Next we save the algorithms of both functions to local variables (compare the end of section 4.2). In line 10 we define a new function via the C++ lambda calculus. The lambda expression captures the local variables we defined by copy and returns upon input the composition of the two algorithms. Since the new algorithm owns shared pointers to the algorithms of the functions to be composed, these algorithms will not be deleted before the new algorithm is destroyed. Finally in line 8 we return a function with the composition of the algorithms as algorithm.

5.3 The class POWERSERIES

The class POWERSERIES is supposed to model formal power series. As such it is a template in the coefficient type, abbreviated by c_t. We want to be able to compute sums and products of power series, therefore the coefficient type needs to have an addition and an multiplication defined. More specific c_t needs to be a unital ring in the sense, that the corresponding standard operators need to be defined (constructors from positive integers, +, $*$, additive inverse).

```
template<class c_t>
class POWERSERIES {
  // members:
    private:
      shared_ptr<const FUNC<c_t(const unsigned int&)>> coeff;
      coeff_fetcher<c_t>* get_coeffs=new coeff_fetcher<c_t>(this);
  // constructors:
    public:
        POWERSERIES(c_t);
        POWERSERIES(FUNC<c_t(const unsigned int&)>);
        POWERSERIES(function<c_t(const unsigned int&)>);
  // standard operators:
    public:
      POWERSERIES& operator = (const POWERSERIES<c_t>&);
      POWERSERIES operator + (const POWERSERIES<c_t>&) const;
      /*...*/
      POWERSERIES operator * (const POWERSERIES<c_t>&) const;
      POWERSERIES operator () (const POWERSERIES<c_t>&) const;
  // member functions:
    public:
      c_t get_coeff(const unsigned int&) const;
      POLY<c_t> cut_of_at(const unsigned int&) const;
      void derive(const unsigned int&);
      void anti_derive(const unsigned int&);
```

We describe its components:

members: The main component of a POWERSERIES is a shared pointer to a sequence, which we represent by an object of type FUNC<c_t(const unsigned int&)>, that is a function from the positive integers to the coefficient type. The second member, an pointer to an object of type 'coeff_fetcher' will be addressed at the end of this subsection, and is merely for performance improvement.

`constructors:` There are three constructors. The first (line 9) takes an object `x` of the coefficient type and returns the power series with first coefficient `x` followed by zeros (This power series represents the constant function which always returns `x`). The second constructor takes what we decided to represent a sequence and returns the corresponding power series. The third one can be directly fed a `C++` lambda expression.

`standard operators:` The standard operators are overloaded for power series. We again emphasise, that the multiplication is the convolution and not point wise. For the composition to be well defined, the first coefficient of the inner power series needs to be zero. Since `c_t` can for example be the type `REAL`, for which a test of equality is not accessible, the composition will not give an warning, if the first coefficient of the inner series is not zero, but the return value will be incorrect.

`member functions:` The member functions should be self explanatory.

The implementations of this class are very straight forward, and can easily be understood from the source code (assuming, that one has read the example implementations of the classes before).

5.3.1 The class `coeff_fetcher`

The class `coeff_fetcher` is a helper class to enable caching of the coefficient values, that also works in constant POWERSERIES. Since the class `coeff_fetcher` is only a helper class and should not be available to the user, it will be put in a anonymous namespace.

To explain the use of the class coeff_fetcher, consider the product `P*Q` of two power series `P` and `Q`. Each time we call `(P*Q).get_coeff(n)`, the convolution of the power series is calculated, which involves `n` multiplications and `n` additions. There are cases, where the same coefficient is needed multiple times, for example if we want to evaluate the powers $\mathtt{P}^i$ of some power series. In this case it would be more efficient, to remember the coefficients and not to recompute them each time.

The above can be implemented by saving a vector `known_coeffs` of pointers to the coefficient type: Each time some, say the `n`-th coefficient is requested, the function `get_coeff` will check, whether `known_coeffs[n]` is defined and not the `NULL` pointer. If it is, it will return the value pointed to. If it is not, it calculate the value and set `known_coeffs[n]` to point to a coefficient type of the result value.

Thus far, this could be implemented in the class `POWERSERIES` itself. There is one problem: It is often reasonable to work with constant power series (for example the power series underlying a `BASE_ANALYTIC` should be constant for the exact same reasons the algorithm of a `FUNC` and the sequence of a `POWERSERIES` is). But in this case, the vector `known_coeffs`, as a member of a constant object, will also be constant. Thus this vector will be stored in a external class `coeff_fetcher`, and be referenced in `POWERSERIES` only by pointer.

We remark, that the class `coeff_fetcher` does have one additional member function `reset()`, to reset the vector of known values to an empty vector. This is for the case, that saving values leads to memory overflow.

6 A class for Taylor series

6.1 The class definition

6.2 The evaluation