Nr.: FIN-001-2018

Multithreading for the expression-dag-based number type Real_algebraic

Martin Wilhelm

Arbeitsgruppe Algorithmische Geometrie

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-001-2018

Multithreading for the expression-dag-based number type Real_algebraic

Martin Wilhelm

Arbeitsgruppe Algorithmische Geometrie

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Wilhelm
Postfach 4120
39016 Magdeburg
E-Mail: martin.wilhelm@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 26.03.2018

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Technical Report

Multithreading for the expression-dag-based number type Real_algebraic

Martin Wilhelm

Department of Simulation and Graphics,
Otto von Guericke University Magdeburg

March 26, 2018

Contents

1. Introduction

Many algorithms, especially in the field of computational geometry, are based on the premise that arithmetic operations are performed exactly. Real machines, however, are based on inexact floating-point arithmetic. Various number types have been developed to close this gap by providing exact computation or, at least, ensuring exact decisions. The second approach is validated through the Exact (Geometric) Computation Paradigm, which states that making exact decisions is sufficient to guarantee the combinatorial correctness of (geometric) algorithms [10].

In this report we describe the implementation of an extension to the exact-decisions number type `Real_algebraic` that enables us to take advantage of multiple processing units. `Real_algebraic` was introduced by Marc Mörig, Ivo Rössling and Stefan Schirra in 2010 with the goal of providing an easily modifiable, efficient exact-decisions number type for algebraic computations [6]. In contrast to previous number types, such as `leda::real` [2] or `CORE::Expr` [5], it is designed as a loose framework for a variety of exchangeable components, so-called policies, making it more flexible and adaptable than its counterparts.

`Real_algebraic` uses a lazy evaluation procedure. When the user invokes an arithmetic operation, it first tries to exactly compute the result locally. If this is not possible, it builds an expression dag. When an expression dag is built, the expression does not get evaluated until its value is requested through a comparison[1]. Then a floating-point filter is invoked and only if the error bound of the filter is not sufficient, the expression gets recomputed with bigfloats up until a sufficiently small error is reached to make an exact decision, i.e., to separate it from zero or to decide that the result is zero through a separation bound (cf. [3, 9]).
A very fundamental policy is the `ExpressionDagPolicy`, which regulates the design and the evaluation process of the underlying expression dag. The main part of this policy is the design of the underlying node class. We describe a new kind of dag node based on the standard `sdag_node`. We include the complete code for our new dag node in this report. However, we will not include any code regarding other policies or `Real_algebraic` itself. For more detailed information on the design and the source code of `Real_algebraic`, see [6, 8].

Our new node type with multithreaded evaluation will be called `mtdag_node`. To parallelize the evaluation we are going to use the C++11 multithreading environment, since it is platform independent, does not require any additional libraries and fits nicely into the programming logic [4]. However, we will as well use the boost thread library to gain some efficiency when implementing a thread manager [1].
In Section 2, we will create a class managing the number of concurrent threads. In Section 3 we describe the new node class.

2. Thread Management

In this section we describe the design and the implementation of a thread manager for the new dag node.

While in theory arbitrarily many processors can be used, a real machine has only a small number of processing units. If the number of threads exceeds the number of processing units, concurrency will be simulated by constantly switching threads. If the number of context switches during a computation is high, this can lead to serious performance issues. Therefore it is important to keep the number of threads created at any point of the computation low.

Since we want to abstract from limitations in the number of threads created, we create a thread pool, called `Thread_manager`, which can receive an arbitrary number of tasks that may potentially be executed in parallel.

Outline

The `Thread_manager` class follows an on-demand approach in the sense that, instead of maintaining a constant sized thread pool, we will create more threads as soon as there are enough unfinished tasks for this to pay off. However, we only create new threads up to a certain threshold to prevent excessive context switching.
This approach has proven to be superior to a permanent thread pool, since the structure of many expression dags is not suited for an efficient use of multiple threads. Even when we have a beneficial structure, for example a perfectly balanced tree, we get faster through eliminating unused threads at the end of the computation, when we have few, but expensive operations near

[1] The evaluation can also be started manually to guarantee a certain error bound, but this is not the use case the number type is designed for.

the root of the tree.

From an implementation point of view, we put some effort in making the thread manager lock-free, i.e., we avoid using mutexes but instead exploit atomic operations when operating on shared data. In particular we partly give up control over the generated threads, which leads to the following condition for usage of `Thread_manager`:

- The *caller*, which is the one providing tasks to the thread manager, must assure that no tasks are open when the destructor of the thread manager is called.

This is a consequence of the fact that we neither have nor want to implement waiting mechanisms inside the class. In addition, it means that the class does not provide any mechanisms to notify the caller, when its tasks are finished.

⟨*Thread_manager.hpp* 3⟩≡

```
#ifndef RA_THREAD_MANAGER_HPP
#define RA_THREAD_MANAGER_HPP

#if (defined RA_CXX11 && defined RA_WITH_BOOST)

⟨tm includes 5.1⟩
⟨tm debug includes 8.4⟩

class Thread_manager{

private:

⟨tm members 4.1⟩
⟨tm thread and task handling 5.2⟩
⟨tm constructors and destructors 8.2⟩
⟨tm guard 4.3⟩

public:

⟨tm add task 6.2⟩
⟨tm instantiation 4.2⟩

};

#endif//RA_CXX11 && RA_WITH_BOOST

#endif//RA_THREAD_MANAGER_HPP
```

For storing the tasks we use a `boost::lockfree::queue` [1], which is why having boost is a hard requirement for using the `Thread_manager` and, consequently, the multithreaded dag node.

Singleton creation

The `Thread_manager` is implemented as a singleton, i.e., there can always be at most one instance of the class which then will be retained until the execution stops. This prevents the class object from getting created again each time a computation is executed.
To achieve this we create our instance in a private static member variable

⟨*tm members* 4.1⟩≡ (3) 4.5▷
```
static Thread_manager *instance;
```

which we then make accessible from outside the class through a public getter function. In the getter function we create an instance of our class, when it is requested for the first time.

⟨*tm instantiation* 4.2⟩≡ (3)
```
static Thread_manager* get_instance() {
  ⟨tm guard instantiation 4.4⟩

  if (instance == nullptr) instance = new Thread_manager();
  return instance;
}
```

We need to make sure that the instance gets deleted when the execution stops. For this we use a (private) guard class, which will delete the instance as soon as its destructor gets called.

⟨*tm guard* 4.3⟩≡ (3)
```
class Thread_manager_guard {
public:
  ~Thread_manager_guard() {
    if (instance != nullptr) {
      delete instance;
      instance = nullptr;
    }
  }
};
```

We then create an instance of this guard class as soon as we get the first call to `get_instance`.

⟨*tm guard instantiation* 4.4⟩≡ (4.2)
```
static Thread_manager_guard guard;
```

By this we have guaranteed that

- There is always at most one instance of `Thread_manager`
- The instance does not get destroyed after each computation.
- The instance gets destroyed when the program execution stops.

Task and thread management

We are going to store incoming tasks in a `boost::lockfree::queue`.

⟨*tm members* 4.1⟩+≡ (3) ◁4.1 5.3▷
```
boost::lockfree::queue<std::function<void()>*> tasks;
```

This is a queue that is both multithread-safe and lock-free, which means it can handle concurrent pop/pop, pop/push and push/push situations without compromising its internal structure *and* without blocking any reader or writer thread in its execution.
The tasks are given as a function object without arguments. So the caller has to bind potential arguments to the function before assigning a task to the `Thread_manager`. We need to include the respective headers for the queue and the functional.

⟨*tm includes* 5.1⟩≡ (3) 5.4▷
```
#include <functional>
#include <boost/lockfree/queue.hpp>
```

Now launched threads will be pulling tasks from the queue until it is empty. When this is the case they are going to automatically destroy themselves. So each new thread will be started with the `execute_tasks` method.

⟨*tm thread and task handling* 5.2⟩≡ (3) 7.2▷
```
void execute_tasks() {
  std::function<void()>* currtask;
  while(tasks.pop(currtask))
  {
    ⟨tm execute task 5.5⟩
  }
  ⟨tm destroy thread 5.6⟩
}
```

The `pop` method of the queue will return false, if it was not able to pop an element, hence the queue was empty at the time the call started. To keep track of the current number of threads, as well as the current number of tasks we store two atomic counters in addition to the queue itself.

⟨*tm members* 4.1⟩+≡ (3) ◁4.5 6.1▷
```
std::atomic<unsigned int> thread_count;
std::atomic<unsigned int> task_count;
```

Note that the queue does not provide any method to check its size, since there is no way of *knowing* the exact size during concurrent pop/push operations. Atomic counters ensure that each increment or decrement operation is executed atomically, which means that no other thread can modify the data while the operation takes place. For native integer data types this should also be done lock-free. We need to include the respective `atomic` header[2].

⟨*tm includes* 5.1⟩+≡ (3) ◁5.1 8.1▷
```
#include <atomic>
```

Now when we execute a task in our `execute_tasks` method, we are going to decrease the task counter. Afterwards we call the associated function and delete the function object.

⟨*tm execute task* 5.5⟩≡ (5.2)
```
--task_count;
currtask->operator()();
delete currtask;
```

When we exit the loop and are about to destroy our current thread, we need to decrement the thread counter. Since this involves some safety measures, it is done in a separate method, we will describe later on.

⟨*tm destroy thread* 5.6⟩≡ (5.2)
```
dec_thread_count();
```

[2]We do not really need to include the header, since it should be included by the lockfree queue as well, but this adds to code stability.

As mentioned in the outline, we will create new threads on demand, up to a certain maximum. For this we define two constants. The constant `MAX_THREADS` determines the number of concurrent threads at which the creation of new threads will stop. Consequently, it can be seen as the maximum number of threads running at any time during the execution[3]. The prefered value of this constant depends heavily on the number of available processing units on the target machine and should be adjusted accordingly. For maximal efficiency, this constant should be about the same size as the number of available processing units.
The second constant, `TASK_THRESHOLD`, determines the number of queued, unfinished tasks, at which a new thread will be created. If this number is too low, this may result in continous creation and destruction of new threads and therefore negatively influence the performance. On the other hand, in our previous tests, we could not see a notable difference in performance when increasing the threshold above five up to one hundred.

⟨*tm members* 4.1⟩+≡ (3) ◁5.3
```
static constexpr int MAX_THREADS = 4;
static constexpr int TASK_THRESHOLD = 5;
```

As we will see soon, due to the non-deterministic execution order, these constants do *neither* imply that there will never be more threads active than `MAX_THREADS`, *nor* that a number of tasks beyond `TASK_THRESHOLD` always leads to the creation of a new thread, *nor* that the `TASK_THRESHOLD` must be reached for a new thread to be created. So these constants should always be seen as soft conditions for task and thread management.

Now that we defined our conditions for thread creation, we describe the methods, in which new threads will be created. This is primarily done, whenever a new task will be added to the thread manager. We define our public interface for task creation as follows

⟨*tm add task* 6.2⟩≡ (3)
```
void add_task(std::function<void()>* pt) {
  ++task_count;
  tasks.push(pt);

  ⟨tm conditioned launch new thread 6.3⟩
}
```

We increase the (atomic) task counter and add the new task to our queue. Note that we first increase the counter and only afterwards store the task in the queue, while in the `execute_tasks` method, we first remove an element from the queue and afterwards decrease the task counter. Therefore the value of the task counter is always greater or equal to the number of tasks stored inside the queue. In particular, `task_count` is zero *only if* the number of tasks inside the queue is zero.

After adding the new task to the queue, we check, whether we should create a new thread

⟨*tm conditioned launch new thread* 6.3⟩≡ (6.2)
```
if (thread_count == 0 || task_count > TASK_THRESHOLD) {
  ⟨tm launch thread if not max 7.1⟩
}
```

Obviously we need to have at least one thread running, whenever there are unfinished tasks inside our queue. Therefore we want to create a new thread, when `thread_count` is zero. Secondly

[3]This is not entirely correct, as described later on

we want to create a new thread, when the number of tasks waiting in our queue gets high and therefore `task_count` surpasses the `TASK_THRESHOLD`. Since the number of tasks inside the queue can be smaller than the value of `task_count`, we cannot guarantee that the threshold is actually reached, when a new thread is created.

Then we create a new thread unless the maximum number of threads is reached.

⟨*tm launch thread if not max* 7.1⟩≡ (6.3)
```
if (thread_count < MAX_THREADS) {
  launch_thread();
}
```

Note that this does not completely prevent the number of threads to surpass `MAX_THREADS`. It is possible that several calls to `add_task` reach this line simultaneously and therefore pass the check before increasing the thread counter.

When we want to create a new thread, we call the `launch_thread` method.

⟨*tm thread and task handling* 5.2⟩+≡ (3) ◁5.2 7.3▷
```
inline void launch_thread() {
  ++thread_count;
  std::thread t(&Thread_manager::execute_tasks,this);
  t.detach();
}
```

In `launch_thread` the value of the thread counter will be increased and a new thread will be created. The newly created thread will then be detached, which means it will run independently from the `Thread_manager` object until the executed method, `execute_tasks` comes to an end. So after leaving this method, we completely give up control over this thread. Therefore it is important (cf. the outline of this section) that the caller ensures that all tasks have finished before the `Thread_manager` object gets destroyed[4].

We increase the thread counter before a new thread gets created, whereas in `execute_tasks` we exit the loop (and therefore effectively kill the thread) before we decrease the thread counter by calling `dec_thread_count`. Therefore, as with `task_count`, the value of `thread_count` is always greater or equal to the number of threads that are (effectively) alive. So whenever `thread_count` is zero, the number of threads alive is zero as well.

However, we cannot reverse the implication. That means, checking whether `thread_count` is greater than zero, as done in `add_task`, is *not* sufficient to ensure that there are any threads alive. Therefore we must check this condition again, after decreasing the thread counter.

⟨*tm thread and task handling* 5.2⟩+≡ (3) ◁7.2
```
inline void dec_thread_count() {
  if (--thread_count == 0 && task_count > 0) {
    launch_thread();
  }
}
```

[4]There is a very unlikely, but (maybe?) possible scenario, in which all tasks are finished, the `Thread_manager` gets destroyed and then one dying thread tries to pop a task from the (already destroyed) queue. Whether this situation can happen, has any implications, and how to prevent this from happening still needs further investigation.

Although we cannot guarantee that `thread_count` is zero when no threads are alive, we know there must be one call to `dec_thread_count`, where `thread_count` reaches zero. Now if there are any tasks left, it may be the case that `add_task` already checked the value of `thread_count` before it got zero. Therefore we create a new thread. Note that this may lead to creating more threads than we actually need. However this may only affect the performance, not the correctness of the execution. We should nevertheless make this situation as unlikely as possible by increasing `thread_count` again as fast as possible after checking that it is zero. Therefore `launch_thread`, where we increase the thread counter, is set `inline`. Also we refrain from checking against `MAX_THREADS`, since having this many threads is quite unlikely (although not impossible) in this situation.

If `task_count` is zero, we do not want to create a new thread (else we would end up in an infinte loop of creating and destroying new threads). However, this additional condition makes our safety check vulnerable. Therefore it is important that `task_count` never underestimates the number of tasks. Because of that the number of tasks inside the queue must be zero and we can safely assume that, if a new task is added, the method `add_task` will start a new thread.

For using `std::thread`, we need to include the respective header

⟨*tm includes* 5.1⟩+≡ (3) ◁5.4
```
#include <thread>
```

Initialization and destruction

Since we want to implement a singleton, constructors and destructors should not be accessible from the outside. So we declare the copy constructor as well as the default destructor private.

⟨*tm constructors and destructors* 8.2⟩≡ (3) 8.3▷
```
Thread_manager(Thread_manager const&){};
~Thread_manager(){};
```

In the constructor, the counters for tasks and threads are initially set to zero, whereas the priority queue will initially reserve space for up to 20 tasks. This space will be automatically extended by the queue, when it is exhausted.

⟨*tm constructors and destructors* 8.2⟩+≡ (3) ◁8.2
```
Thread_manager():tasks(20),thread_count(0),task_count(0){
  assert(tasks.is_lock_free());
  assert(thread_count.is_lock_free());
  assert(task_count.is_lock_free());
}
```

We also check on construction, whether the implementation of the queue and the atomic variables is lock-free. These are only soft assertions, i.e., if these assertions fail it will not affect the correctness of the implementation. However, the performance may be drastically affected, which is why we decided to place these assertions. To be able to use `assert`, we need to include its header.

⟨*tm debug includes* 8.4⟩≡ (3)
```
#include <cassert>
```

Finally we need to initialize the `instance` pointer. Since it is static, it cannot be initialized in the header. So we need to create a source file, where we initialize `instance` to be a null pointer.

⟨*Thread_manager.cpp* 9⟩≡
```
#ifndef RA_THREAD_MANAGER_CPP
#define RA_THREAD_MANAGER_CPP

#ifdef RA_CXX11

#include "Thread_manager.hpp"

Thread_manager* Thread_manager::instance = nullptr;

#endif//RA_CXX11
#endif//RA_THREAD_MANAGER_CPP
```

3. DAG Node Implementation

The goal of the following section is to parallelize the evaluation of the expression dag based on the evaluation model of the `sdag_node`. We will focus on parallelizing the accuracy-driven computation, since the main part of the complexity in the form of bigfloat operations lies there.

Outline

In 2015 Mörig and Schirra suggested using a topological evaluation order instead of the recursive procedure used in `sdag_node` for the accuracy-driven evaluation [7]. This does not only fix some performance issues, it also enables us to parallelize the evaluation of nodes that are not dependent on each other, i.e., those nodes that are not connected through a directed path. We therefore assume that the evaluation is done in topological order. However we do only comment on the parts of the evaluation that must be adjusted for multithreaded evaluation. The remaining code can be found in the appendix.

3.1. Ensuring Multithread-safety

Before we can parallelize the evaluation of independent nodes, we have to make sure that no data conflicts occur between such nodes. In particular we must remove or guard each write access to a child node during recomputation. Otherwise multiple, independent parents of the same node could get in conflict with each other.
There are two situations in which a non-read-only access to a child node can occur. These are

1. Reference handling
2. Computation of an algebraic degree bound for the node

In the first case a parent node may delete its reference to a child node during computation, when it is converted to a bigfloat. We will handle this by ensuring that no access to the reference counter can corrupt it.
The second case is more troublesome. After each evaluation a separation bound for the current node is computed to decide whether the node evaluates to zero. For this we need to know an upper bound on the algebraic degree of the current node, which also may change (get smaller) during the computation if roots cancel out or can be computed exactly.
In previous dag node variants the algebraic degree bound of an expression gets recomputed top-down recursively *every time* a node gets recomputed. During this process, a temporary status

member variable is used to indicate whether the node was already visited. This variable may get corrupted when multiple parents try to recompute their algebraic degree bound simultaneously. We solve this conflict by replacing the top-down computation by an bottom-up approach, in which each parent only needs read-only access to child node data. In addition to making it multithread safe, this also replaces the previously quadratic runtime of the degree computation by an almost linear one[5].

Reference handling

As mentioned before, we need to ensure that simultaneous write access to the reference counter is handled correctly. This can be done by making the counter atomic.

⟨*reference handling* 10.1⟩≡ (21) 10.2▷

```
std::atomic<unsigned int> count;
```

By this each increment and decrement operation, *including the return of its value*, will be executed atomically. Since the used access methods consist of only one such operation each and `ref_minus` returns by value, we can use the same code as before for them.

⟨*reference handling* 10.1⟩+≡ (21) ◁10.1

```
inline void ref_plus(){ ++count; }
inline unsigned int ref_minus(){ return --count; }
```

Making the counter atomic instead of using a lock mechanism is quite convenient and saves some overhead compared to maintaining a mutex. However, since most calls to `ref_plus` and `ref_minus` do not need to be multithread safe, it might actually be more efficient to use a lock in the few situations, when a node is converted to a bigfloat during accuracy-driven computation. We chose the atomic variant over the use of mutexes, because in the current implementation we managed to avoid maintaining a mutex for each node and the overhead of atomic counters over standard counters is negligible compared to the rest of the computation. If in future implementations a need for mutexes emerges independently from reference handling, one may consider replacing the usage of atomic by mutexes.

Algebraic degree bound computation

In the following paragraphs we will replace the top-down approach in degree computation by a bottom-up approach. An upper bound for the algebraic degree of an expression in the form of an expression dag can be found by multiplying the degree of all of its nodes containing a root operation. In a tree, this can be done by recursively multiplying the degree bounds of a node's children. However, in a dag this may result in factoring in nodes with multiple references more than once and consequently may lead to a strong overestimation of the algebraic degree. So we must take this into consideration.

We develop our algorithm around the premise that usually the number of nodes, which influence the algebraic degree of an expression, is fairly small. If this would not be the case, then the separation bound we get would get very large, probably too large for any practical application, and therefore the cost associated with the evaluation would outweigh any cost associated with the degree bound computation.
Based on this premise, we (temporarily) store the algebraic degree bound of a node during evaluation by splitting it up into a variable called `unique_algebraic_degree` and a collection of descendants with degree greater than one.

[5] Due to the used data structures, the runtime of the implementation will not be linear in theory, but for almost all practical purposes.

⟨*algebraic degree computation members* 11.1⟩≡ (21) 11.3▷
```
Exponent unique_algebraic_degree;
std::vector<SelfType const*> vardeg_nodes;
```

In `unique_algebraic_degree` the part of the algebraic degree bound will be stored, which is unique to this node in the sense that it does not contain the degree bounds of any descendants which may be used again by one of its parents. Every descendant with a degree greater than one that may be used again is stored in `vardeg_nodes`.
The operations we will need for our collection are union and find, both of which are inefficient in an array-like data structure like a vector. Nevertheless we still use a vector, since we expect the number of elements inside the vector to be very small, since these elements must have both

- A `unique_degree` larger than one and
- More than one direct parent.

So presumably the number of elements in the vector will be almost always smaller than ten, in many cases even zero or one. For collections of this size the overhead of more complex data structures is not worth the performance gain, if there even is any gain at all.

The bound for the algebraic degree will then be computed by multiplying the unique parts of the node itself and of all nodes in `vardeg_nodes`.

⟨*separation bound computation compute degree* 11.2⟩≡ (13.3) 12.1▷
```
Exponent get_algebraic_degree() const{

  assert(unique_algebraic_degree >= 1);

  Exponent algebraic_degree = unique_algebraic_degree;
  for (auto node : vardeg_nodes) {
    algebraic_degree *= node->get_algebraic_degree();
  }

  return algebraic_degree;
}
```

To identify which nodes have multiple parents, we use a temporary variable, which will be set during the preprocessing, i.e., the topological sorting.

⟨*algebraic degree computation members* 11.1⟩+≡ (21) ◁11.1 11.4▷
```
bool has_multiple_references;
```

During preprocessing we will also reset all temporary parameters. The `unique_algebraic_degree` will be set to the degree of the node, which is d for the d-th root and 1 otherwise.

⟨*algebraic degree computation members* 11.1⟩+≡ (21) ◁11.3
```
void reset_algebraic_degree_parameters() {
  unique_algebraic_degree = degree();
  vardeg_nodes.clear();
  has_multiple_references = false;
}
```

Now during postprocessing of the computation step we compute the new degree bound for the node by factoring in its child nodes.

⟨*separation bound computation compute degree* 11.2⟩+≡ (13.3) ◁11.2 12.2▷
```
void compute_degree_bound(){
  assert(unique_algebraic_degree == degree());
  alg_degree_factor_in(X);
  alg_degree_factor_in(Y);
}
```

If a child exists, we need to decide, whether it has multiple parents and needs to be stored in `vardeg_nodes`. Either way we must merge the `vardeg_nodes` of the child with our own collection (which will be initially empty).

⟨*separation bound computation compute degree* 11.2⟩+≡ (13.3) ◁12.1 12.5▷
```
void alg_degree_factor_in(SelfType const* child){
  if (child) {
    ⟨factor in child unique degree 12.3⟩
    ⟨merge child vardeg nodes 12.4⟩
  }
}
```

The only situation in which we must consider a child's unique degree at all is if it is greater than one. If the child node has multiple parents, we must make sure that it is not counted more than once and therefore add it to our collection, else it cannot occur anywhere else and therefore we can include it in the node's unique degree.

⟨*factor in child unique degree* 12.3⟩≡ (12.2)
```
if (child->unique_algebraic_degree > 1) {
  if (child->has_multiple_references) {
    add_var_deg_node(child);
  } else {
    unique_algebraic_degree *= child->unique_algebraic_degree;
  }
}
```

If the child has any `vardeg_nodes` itself, we must merge them with our own, which is, we effectively merge the sets of `vardeg_nodes` of our left and right child. By ensuring that no node appears twice in our collection, we also ensure that none of those degrees contributes to our final algebraic degree bound more than once.

⟨*merge child vardeg nodes* 12.4⟩≡ (12.2)
```
for (auto node : child->vardeg_nodes) {
  add_var_deg_node(node);
}
```

Since we do not have a set union operation in a vector, we must iterate through all nodes in the first collection and insert them into the second one. So we search for each node in the parent's vector and, if it is not already there, we insert it at the end.

⟨*separation bound computation compute degree* 11.2⟩+≡ (13.3) ◁12.2
```
inline void add_var_deg_node(SelfType const* node) {
  if (std::find(vardeg_nodes.begin(),vardeg_nodes.end(),node) == vardeg_nodes.end()) {
    vardeg_nodes.push_back(node);
  }
}
```

Separation bound computation

We need the algebraic degree bound for the computation of a separation bound. To be able to enforce const correctness, which is very useful in a multithreaded environment, we first introduce a new getter method for the separation bound object.

⟨*separation bound getter and setter* 13.1⟩≡ (24.4) 25.2 ▷
```
inline SeparationBound const& get_sep_bd() const { return node_data->sep_bd; }
```

Then we use this getter and the newly defined method `get_algebraic_degree` to call the separation bound computation.

⟨*separation bound computation wrapper* 13.2⟩≡ (13.3)
```
inline Exponent sep_bound() const{
  return get_sep_bd().bound(get_algebraic_degree());
}
```

It is important that `compute_degree_bound` is called before `sep_bound`, as well as that all children of the current node have already been recomputed. Since there is no way of detecting whether the computation has already been executed, in some cases this leads to errors, more precisely to underestimations of the algebraic degree, that are hard to find.
We summarize the methods in a separation bound computation chunk.

⟨*separation bound computation* 13.3⟩≡ (21)
```
⟨separation bound computation wrapper 13.2⟩
⟨separation bound computation compute degree 11.2⟩
⟨separation bound concrete computation 29.2⟩
```

The separation bound is used in the method `fixup_post_recompute` directly after the recomputation of a node's approximation to decide whether the value of the node is zero (see also the implementation of `recompute_op` in Section 3.2). We adjust this method to account for the initialization of our new algebraic degree bound computation parameters.

⟨*adc fixup* 13.4⟩≡ (33.2)
```
void fixup_post_recompute(){

  if(exact()){
    convert_to_bigfloat();
    ⟨reset parameters 13.5⟩
  } else {
    ⟨check if node is exactly zero 14.1⟩
  }

  ⟨try to improve filter policy 31.1⟩
}
```

Whenever the node is converted to a bigfloat during the computation, we have to reset our computation parameters. This can happen whenever an exact value is detected, i.e., when the approximation is exact or when the node can be identified as exactly zero. As before, we also reset the separation bound.

⟨*reset parameters* 13.5⟩≡ (13.4 14.1)
```
init_sep_bd();
reset_algebraic_degree_parameters();
```

If the approximation is not exact, we check for the separation bound. Before that, we have to compute the bound for the algebraic degree. Note that, since this is not done recursively, we must ensure that the `compute_degree_bound` method is called in every (non-exact) node, irrespective of whether the separation bound actually gets computed or not.

⟨*check if node is exactly zero* 14.1⟩≡ (13.4)
```
init_sep_bd();
compute_degree_bound();

if( is_zero() ||
    floor_log2_approx() <= ceil_log2_error() ){
  Exponent q = get_requested_error();

  if(q + 1 < sep_bound()){
    zeroize();
    convert_to_bigfloat();
    ⟨reset parameters 13.5⟩
  }
}
```

If the node can be evaluated to zero and therefore converted to a bigfloat, we again have to reset our computation parameters.

3.2. Parallelizing the Evaluation

After making the node class multithread-safe, we can finally parallelize the evaluation. For this we will use the thread manager as described in Section 2.

Parallel Recomputation

We will only focus on the recomputation of the approximation. In particular, we will not try to parallelize the topological sorting or the error propagation.

⟨*adc wrapper* 14.2⟩≡ (33.2) 17.1 ▷
```
void guarantee_bound_two_to(const Exponent& q){
  if(exact() || ceil_log2_error() <= q) return;

  std::vector<SelfType*> order;
  topsort_visit(this,order);

  requested_error() = q;

  ⟨propagate error topologically descending 33.3⟩
  ⟨recompute approximation thread based 16.2⟩
}
```

However, we still have to adjust the implementation of the topological sorting to fit our needs (see 'Preprocessing'). Our strategy for recomputation is as follows

1. Recompute all leaf nodes and notify their parents when the computation is finished.
2. Start the computation of an inner node as soon as all its children have sent a notification.
3. When the root node has been recomputed, wake up the main thread and continue its execution.

So in this strategy, the responsibility of starting the recomputation of inner nodes can be shifted from the main thread to the nodes and is therefore decentralized. This has several advantages:

1. There is no need for an expensive communication strategy between the main thread and the subthreads, except for the notification by the root node.
2. We can start new tasks on demand and therefore whenever a task is started, we can guarantee that its requirements are satisfied. Consequently, we never have to wait for any requirement during execution.
3. This allows us to use a notification mechanism, which can be executed lock-free[6], except for the communication between root node and main thread.
4. We do not depend on the thread manager to detect, when the computation is finished.

On the downside, each node needs to know its parents in the current part of a (possibly larger) dag, in which the recomputation is executed. Therefore we need to assign all current parents to a node during preparation, i.e., during the topological sorting. We save them in a local vector.

⟨*parent handling* 15.1⟩≡ (21) 15.2▷
```
std::vector<SelfType*> parents;
```

Furthermore each parent node needs to know how many dependencies it has, i.e., how many notifications are necessary until its recomputation can be started. This number may be different from the number of children as we will see later, since not all nodes need recomputation.

⟨*parent handling* 15.1⟩+≡ (21) ◁15.1 15.3▷
```
std::atomic<int> dependency_count;
```

It is very important that `dependency_count` is atomic, since its children may simultaneously try to notify their parent and therefore decrease its dependency counter. This issue will become apparent soon.
To ensure that the number of notifying children fits the number of dependencies, we will put their initialization together in one method.

⟨*parent handling* 15.1⟩+≡ (21) ◁15.2 15.4▷
```
inline void register_parent(SelfType* p) {
  parents.push_back(p);
  ++(p->dependency_count);
}
```

We will now get to the core method that decides when to launch a new task.

⟨*parent handling* 15.1⟩+≡ (21) ◁15.3
```
void try_recomputation () {
  if (dependency_count-- == 0)  {
    ⟨start new task 16.1⟩
  }
}
```

[6]It means without the need for semaphores or other locking mechanisms.

Calling the method `try_recomputation` equates to notifying the node that one of its dependencies has been fulfilled. If there are no dependencies left, its recomputation task will be started. We decrease the dependency counter *after* checking whether it is zero, since this method will be called once from the main thread for each node, to start those that have no dependencies. So for each node this method will be called exactly `dependency_count+1` times and the task will be started during the last call[7], although we do not know which call will be the last.

It is necessary for the decrease operation and the check to happen atomically, i.e., for `dependency_count` to be atomic. If the counter would be decreased in a separate operation, e.g. in an else block, it would be possible that the new task is never started. This happens, if the following sequence of events occurs:

1. child a and child b want to notify their parent node with `dependency_count` 1
2. child a checks `dependency_count`, which is 1
3. child b checks `dependency_count`, which is 1
4. child a decreases `dependency_count` to 0
5. child b decreases `dependency_count` to -1

To make things worse, accessing any member variable after the check may lead the program to crash or, even worse, produce wrong results, since the execution may have already stopped at this point or a new computation may have started.

It should also be noted that the atomic keyword guarantees that the return value of a pre- or post-decrement operation is given by value. So even if another decrement occurs in between the previous post-decrement and the evaluation of the equals operator, it will not affect the result of the evaluation[8].

We start a new task by binding the node to the `recompute_op` function (which we describe below) and passing it to the `Thread_manager`.

⟨*start new task* 16.1⟩≡ (15.4)
```
Thread_manager::get_instance()->add_task(
  new std::function<void()>(std::bind(recompute_op,this))
);
```

As mentioned before, we try to start the computation of each node once inside the main thread.

⟨*recompute approximation thread based* 16.2⟩≡ (14.2)
```
⟨prepare synchronization 18.4⟩

for(auto node : order) {
  node->try_recomputation();
}
⟨synchronize threads 19.1⟩
```

[7] This refers to the call, which decreased the counter last, not necessarily the call, which has been started last
[8] Admittedly, return by reference should only be possible to occur for pre-decrement anyway.

This will be the starting point for each node that does not have any dependencies. It can also be potentially the starting point of nodes, whose dependencies were resolved before the main loop reached them. After notifying each node once, we will pause the execution until all tasks are finished. This will be described in more detail in the paragraph 'Synchronization'.

The `recompute_op` method takes a node and recomputes its approximation such that the error will be smaller than the requested error, assuming that the child nodes have already been recomputed with appropriate error bounds. Since this method will be passed to the thread manager, it must be static.
The recomputation itself happens in the same way as without parallelization. However, after the node is recomputed, it must now send a notification to its parents or, if it is the root node, to the main thread. Also note that we already slightly modified the fixup method in Section 3.1 to implement the new algebraic degree bound computation strategy.

⟨*adc wrapper* 14.2⟩+≡ (33.2) ◁14.2
```
static void recompute_op(SelfType* node) {

  assert(node->contype() != DOUBLE);
  assert(node->contype() != BIGFLOAT);

  switch(node->contype()) {
    case NEGATION:       node->recompute_neg();  break;
    case ROOT:           node->recompute_root(); break;
    case ADDITION:       node->recompute_add();  break;
    case SUBTRACTION:    node->recompute_sub();  break;
    case MULTIPLICATION: node->recompute_mul();  break;
    case DIVISION:       node->recompute_div();  break;
  }

  node->fixup_post_recompute();
  node->topsort_status() = CLEARED;

  ⟨send notifications 17.2⟩
}
```

The root node can be identified by checking, whether its `parents` vector is empty. Since we only assign parents relevant to the recomputation, the root node will never have parents. On the contrary, each other node must have at least one parent, since otherwise it would not influence the value of the root node and therefore would not have been recomputed in the first place.

⟨*send notifications* 17.2⟩≡ (17.1)
```
if (node->parents.empty()) {
  ⟨synchronize with main thread 19.2⟩
} else {
  ⟨notify parents 17.3⟩
}
```

If the node is not the root, we notify its parents that it is ready.

⟨*notify parents* 17.3⟩≡ (17.2)
```
for (auto p: node->parents) {
  p->try_recomputation();
}
```

As before there cannot be any statement after the last call to `try_recomputation`, since from this point on we have no guarantees about the context anymore. The node could be deleted or another computation could have started after notifying the last parent.

Synchronization

While working on the tasks, we need our main thread to pause its execution. Unfortunately we cannot use this thread to compute tasks itself due to our decentralized strategy and the strict separation between the computation and the task manger. So we must wait until the root node notifies the main thread that its recomputation has finished. We do this by using a *mutex* in combination with a *condition variable*.

⟨*threading members* 18.1⟩≡ (21) 18.2▷
```
static std::mutex sync_mutex;
static std::condition_variable sync_cv;
```

Since we need only one of them, we make them static. We also need a boolean, indicating when the computation is finished.

⟨*threading members* 18.1⟩+≡ (21) ◁18.1
```
static bool computation_finished;
```

Since they are static template members, we must initialize all three variables outside the class declaration in the header file.

⟨*static member definition* 18.3⟩≡ (21)
```
template <class P> std::mutex mtdag_node<P>::sync_mutex;
template <class P> std::condition_variable mtdag_node<P>::sync_cv;
template <class P> bool mtdag_node<P>::computation_finished = false;
```

While a mutex is simply a semaphore, which can be locked atomically and blocks all other lock attempts until the lock is freed, a condition variable functions as a permanent[9] block until it receives a notification from another thread and its condition is fulfilled. It is designed such that, while a thread is blocked through a condition variable, it consumes minimal time during context switches.

A condition variable always depends on an underlying mutex, which it will lock as soon as it wakes up to test for its condition. If the condition is not fulfilled yet, it will go to sleep again and unlock the mutex. This check also happens before waiting for the first notification, which is why the mutex needs to be locked before starting the waiting process.

So before starting any tasks, we set our condition, the boolean `computation_finished`, to `false`.

⟨*prepare synchronization* 18.4⟩≡ (16.2)
```
computation_finished = false;
```

A `unique_lock` immediately locks the corresponding mutex on construction and unlocks it again either when its `unlock` method is called, or when it gets destructed. After we tried to recompute each node once, we lock the synchronization mutex with a `unique_lock` and start the waiting process.

[9] Aside from so-called 'spurious wakeups'.

⟨*synchronize threads* 19.1⟩≡ (16.2)

```
std::unique_lock<std::mutex> lk(sync_mutex);
sync_cv.wait(lk,[this](){ return computation_finished; });
```

The condition variable lets the thread wait until it gets notified and its condition is `true`. While it is waiting, `sync_cv` unlocks the mutex. If it wakes up, the mutex will be locked again. Note that a wakeup can also occur 'spuriously', without any notification from other threads. Therefore it is important *not* to set `computation_finished` to true before the root node has finished its computation.

Finally after recomputing the root, we set `computation_finished` to `true` and notify the condition variable.

⟨*synchronize with main thread* 19.2⟩≡ (17.2)

```
std::lock_guard<std::mutex> sync_lk(node->sync_mutex);

node->computation_finished = true;
node->sync_cv.notify_one();
```

A `lock_guard` is a simple lock, which locks the mutex on creation and only unlocks on destruction. This makes it a bit more efficient than the previously used `unique_lock`. We must lock `sync_mutex` before changing the condition and notifying `sync_cv` to prevent the following sequence of events from happening.

1. `computation_finished` is set to false
2. The computation gets finished
3. `sync_cv` checks `computation_finished`
4. `computation_finished` is set to true
5. `sync_cv.notify_one()` is called
6. `sync_cv` goes to sleep

By using the lock, the thread computing the root node must wait until `sync_cv` goes to sleep and unlocks the mutex, before notifying it[10]. Similarly, the condition variable must wait until the other threads `lock_guard` has been destroyed, before it can check its condition again.

Preprocessing

In this section, we have a look at the preprocessing that is necessary before each recomputation of the dag. Before starting the recomputation, a topological order must be determined and the temporary parameters must be initialized. We combine both steps in the method `topsort_visit`.

[10] Except when it is called before the `unique_lock` is created. Then the main thread will never start waiting, since the condition is fulfilled from the beginning.

⟨*topological sorting* 20.1⟩≡ (21)

```
static void topsort_visit(SelfType* node,
                          std::vector<SelfType*>& order){

  if(node->topsort_status() != VISITED){

    ⟨reset temporary parameters 20.2⟩
    ⟨topsort visit children 20.3⟩
    ⟨check exactness and add to order 20.4⟩

    node->topsort_status() = VISITED;
  } else {
    node->has_multiple_references = true;
  }
}
```

The variable `topsort_status` is used to determine whether the node was already visited. In addition, if it was visited before, we now set `has_multiple_references` to `true`, which is used in computing the algebraic degree bound. Also, at the first visit, we reset all temporary parameters we will need in either the algebraic degree computation, or for maintaining the parents.

⟨*reset temporary parameters* 20.2⟩≡ (20.1)

```
node->reset_algebraic_degree_parameters();
node->parents.clear();
node->dependency_count = 0;
```

When the node has any children, we will also register it as a parent for them, if they are not exact. Otherwise they do not need to be recomputed and therefore will not be added to the list. So an inner node can have no dependencies, if all of its children are exact.

⟨*topsort visit children* 20.3⟩≡ (20.1)

```
if(node->X) {
  topsort_visit(node->X,order);
  if (!node->X->exact()) node->X->register_parent(node);
}
if(node->Y) {
  topsort_visit(node->Y,order);
  if (!node->Y->exact()) node->Y->register_parent(node);
}
```

Note that it is important to first visit the children and then call `register_parent`, since `parents` will be reset at the first call of `topsort_visit`. Finally, all nodes which are not exact yet are added to `order` for recomputation. If the node is exact, it does not need any recomputation and therefore will be treated as read-only during the evaluation process, so we need to initialize the separation bound right away.

⟨*check exactness and add to order* 20.4⟩≡ (20.1)

```
if(node->exact()) {
  node->init_sep_bd();
} else {
  order.push_back(node);
}
```

3.3. Class overview

It follows the class definition. In contrast to `sdag_node` we do not use the custom memory management of `Real_algebraic`, since this is not guaranteed to be multithread-safe[11]. We require `boost` for using this class because the `Thread_manager` class relies on `boost`, although it is not used directly in `mtdag_node`. So with an alternate thread manager, we could get rid of this requirement.

⟨*mtdag_node.hpp* 21⟩≡

```
#ifndef RA_MTDAG_NODE_HPP
#define RA_MTDAG_NODE_HPP

#if (defined RA_CXX11 && defined RA_WITH_BOOST)

⟨includes 23.1⟩

template <class Policies>
class mtdag_node{
public:

  typedef mtdag_node<Policies> SelfType;

  ⟨typedefs and enums 23.2⟩

  ⟨node data declaration 24.1⟩
  ⟨node data getter and setter 24.4⟩

  ⟨initialize bigfloat data 25.5⟩
  ⟨topological sorting 20.1⟩

  ⟨members 23.3⟩
  ⟨parent handling 15.1⟩
  ⟨reference handling 10.1⟩
  ⟨node type 23.4⟩
  ⟨threading members 18.1⟩
  ⟨algebraic degree computation members 11.1⟩

  ⟨constructors 27.1⟩
  ⟨destructor 28.2⟩

  ⟨adjust hardware fp interval 26.2⟩
  ⟨fixed precision computation 30.3⟩

  ⟨sign computation 29.3⟩

  ⟨guaranteeing error 28.3⟩

  ⟨separation bound computation 13.3⟩
  ⟨accuracy driven computation 33.2⟩

  ⟨bigfloat conversion 26.1⟩
};

⟨static member definition 18.3⟩
```

[11] The same is true for the bigfloat data, whose definition can be found in the appendix.

```
#endif//RA_CXX11 && RA_WITH_BOOST
#endif//RA_MTDAG_NODE_HPP
```

References

[1] T. Blechmann. Boost.lockfree library documentation. `http://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html`, 2008-2011. Accessed: 2018-02-26.

[2] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry, Miami Beach, Florida, USA, June 13-16, 1999*, pages 341–350, 1999.

[3] S. Fortune and C. J. V. Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the Ninth Annual Symposium on Computational GeometrySan Diego, CA, USA, May 19-21, 1993*, pages 163–172, 1993.

[4] Standard for Programming Language C++. Standard, International Organization for Standardization, Sept. 2011.

[5] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry, Miami Beach, Florida, USA, June 13-16, 1999*, pages 351–359, 1999.

[6] M. Mörig, I. Rössling, and S. Schirra. On design and implementation of a generic number type for real algebraic number computations based on expression dags. *Mathematics in Computer Science*, 4(4):539–556, 2010.

[7] M. Mörig and S. Schirra. Precision-driven computation in the evaluation of expression-dags with common subexpressions: Problems and solutions. In *6th International Conference on Mathematical Aspects of Computer and Information Sciences, MACIS*, pages 451–465, 2015.

[8] M. Mörig. *Algorithm Engineering for Expression Dag Based Number Types*. Dissertation, Otto-von-Guericke Universität Magdeburg, 2015.

[9] S. Schirra. Much ado about zero. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, pages 408–421, 2009.

[10] C. Yap. Towards exact geometric computation. *Comput. Geom.*, 7:3–23, 1997.

A. Appendix: Remaining code segments

Class body

⟨*includes* 23.1⟩≡ (21)

```
#include "floating_point_basics.hpp"
#include "memory_allocation.hpp"
#include "error_handling.hpp"
#include "bits_and_pieces.hpp"
#include <cstdlib>
#include <vector>

#include "Thread_manager.hpp"
#include <mutex>
#include <condition_variable>
```

⟨*typedefs and enums* 23.2⟩≡ (21)

```
typedef typename Policies::FilterPolicy         FilterPolicy;
typedef typename Policies::ApproximationPolicy  ApproximationPolicy;
typedef typename Policies::SeparationBound      SeparationBound;
typedef ApproximationPolicy        AP;
typedef typename AP::Approximation Approximation;
typedef typename AP::Exponent      Exponent;
typedef typename AP::Precision     Precision;

enum statustype{CLEARED, VISITED};

enum constructortype{DOUBLE, BIGFLOAT,
                     NEGATION, ROOT,
                     ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION};
```

⟨*members* 23.3⟩≡ (21)

```
FilterPolicy I;

Node_data *node_data;
SelfType *X, *Y;
```

⟨*node type* 23.4⟩≡ (21)

```
unsigned int node_type;

static inline unsigned int
make_nt(unsigned int degree, unsigned int type){
  return (degree << 4) | type;
}

inline int degree() const{ return (node_type >> 4);}
inline int contype() const{ return (node_type & 15);}
```

Bigfloat data

⟨*node data declaration* 24.1⟩≡ (21)

```
struct Node_data{

  ⟨node data members 24.2⟩
  ⟨node data constructors 24.3⟩

private:

  ⟨ceil log2 double 25.4⟩
};
```

⟨*node data members* 24.2⟩≡ (24.1)

```
Approximation approx;
Approximation error;

SeparationBound sep_bd;

Exponent requested_error;
statustype topsort_status;
```

⟨*node data constructors* 24.3⟩≡ (24.1)

```
Node_data():topsort_status(CLEARED){}

Node_data(const double x,const double e)
       :approx(x),error(e),requested_error(e==0?0:ceil_log2(e)),
        topsort_status(CLEARED){}

Node_data(const Approximation& x,
          const double e)
       :approx(x),error(e),requested_error(e==0?0:ceil_log2(e)),
        topsort_status(CLEARED){}

Node_data(const Approximation& x,
          const Approximation& e)
       :approx(x),error(e),
        requested_error(AP::sign(error)==0?0:AP::ceil_log2(error)),
        topsort_status(CLEARED){}
```

⟨*node data getter and setter* 24.4⟩≡ (21)

⟨*bigfloat getter and setter* 25.1⟩
⟨*separation bound getter and setter* 13.1⟩
⟨*topological evaluation getter and setter* 25.3⟩

⟨*bigfloat getter and setter* 25.1⟩≡ (24.4)

```
inline Approximation& approx(){  return node_data->approx;  }
inline Approximation const& get_approx() const {  return node_data->approx;  }
inline Exponent ceil_log2_approx() const { return AP::ceil_log2(get_approx()); }
inline Exponent floor_log2_approx() const { return AP::floor_log2(get_approx()); }

inline Approximation& error(){  return node_data->error;  }
inline Approximation const& get_error() const { return node_data->error; }
inline Exponent ceil_log2_error() const { return AP::ceil_log2(get_error()); }

inline bool exact() const { return !static_cast<bool>(AP::sign(get_error())); }
inline int sign() const { return AP::sign(get_approx()); }
inline bool is_zero() const { return sign() == 0; }
```

⟨*separation bound getter and setter* 13.1⟩+≡ (24.4) ◁13.1

```
inline SeparationBound& sep_bd(){ return node_data->sep_bd; }
```

⟨*topological evaluation getter and setter* 25.3⟩≡ (24.4)

```
inline Exponent& requested_error(){ return node_data->requested_error; }
inline Exponent const& get_requested_error() const { return node_data->requested_error; }

inline statustype& topsort_status(){ return node_data->topsort_status; }
```

⟨*ceil log2 double* 25.4⟩≡ (24.1)

```
static int ceil_log2(const double d) {
  int k;
  std::frexp(d,&k);
  return k;
}
```

⟨*initialize bigfloat data* 25.5⟩≡ (21)

```
void init_node_data() {
  if (node_data) return;

  if (X) X->init_node_data();
  if (Y) Y->init_node_data();

  if(contype() == DOUBLE){
    node_data = new Node_data(I.get_point(),0.0);
  } else if (I.is_finite()){
    node_data = new Node_data(I.get_median(),I.get_radius());
    if (exact()) convert_to_bigfloat();
  } else {
    node_data = new Node_data();
    compute_op(AP::initial_prec());
  }
}
```

Convenience methods

⟨*bigfloat conversion* 26.1⟩≡ (21)

```
inline void zeroize() {
  I = FilterPolicy(0.0);
  AP::zeroize(approx());
  AP::zeroize(error());
}

inline void convert_to_bigfloat() {
  if (X && (X->ref_minus() == 0)) delete X;
  if (Y && (Y->ref_minus() == 0)) delete Y;
  X = Y = 0;
  node_type = make_nt(1,BIGFLOAT);
}
```

⟨*adjust hardware fp interval* 26.2⟩≡ (21)

```
void adjust_interval(){
  AP::to_interval(I,get_approx());

  if(!exact()){
    const double rad = AP::to_double(get_error(),AP::round_up());
    FilterPolicy err;
    err.set_median_and_radius(0.0,rad);
    I += err;
  }
}
```

⟨*adc auxillary functions* 26.3⟩≡ (33.2)

```
inline Exponent ceil_log2_high() const {
  if (is_zero()) return ceil_log2_error();
  if (exact()) return ceil_log2_approx();

  return static_cast<Exponent>(1) + std::max(ceil_log2_approx(),ceil_log2_error());
}

inline Exponent floor_log2_low() const {
  assert(!is_zero());
  if(exact()) return floor_log2_approx();

  assert(ceil_log2_error() < floor_log2_approx());
  return floor_log2_approx()-static_cast<Exponent>(1);
}
```

Constructors and destructors

⟨*constructors* 27.1⟩≡ (21)

```
mtdag_node(const FilterPolicy& i,Int_to_type<BIGFLOAT>)
:I(i),node_data(new Node_data()),X(0),Y(0),count(1),node_type(make_nt(1,BIGFLOAT)){}

mtdag_node(const double x)
:I(x),node_data(0),X(0),Y(0),count(1),node_type(make_nt(1,DOUBLE)){
  assert(ra_isfinite(x));
}

mtdag_node(const Approximation& x)
:node_data(new Node_data(x,0.0)),X(0),Y(0),count(1),node_type(make_nt(1,BIGFLOAT)) {
  adjust_interval();
}

mtdag_node(mtdag_node* a,mtdag_node* b,Int_to_type<ADDITION>)
:I( a->I + b->I ),node_data(0),X(a),Y(b),count(1),node_type(make_nt(1,ADDITION)){
  ⟨increase operands reference counter 27.2⟩
}

mtdag_node(mtdag_node* a,mtdag_node* b,Int_to_type<SUBTRACTION>)
:I( a->I - b->I ),node_data(0),X(a),Y(b),count(1),node_type(make_nt(1,SUBTRACTION)){
  ⟨increase operands reference counter 27.2⟩
}

mtdag_node(mtdag_node* a,mtdag_node* b,Int_to_type<MULTIPLICATION>)
:I( a->I * b->I ),node_data(0),X(a),Y(b),count(1),node_type(make_nt(1,MULTIPLICATION)){
  ⟨increase operands reference counter 27.2⟩
}

mtdag_node(mtdag_node* a,mtdag_node* b,Int_to_type<DIVISION>)
:I( a->I / b->I ),node_data(0),X(a),Y(b),count(1),node_type(make_nt(1,DIVISION)){
  ⟨increase operands reference counter 27.2⟩
}

mtdag_node(mtdag_node* a,const constructortype con)
:I(-(a->I)),node_data(0),X(a),Y(0),count(1),node_type(make_nt(1,con)){
  assert(contype() == NEGATION);
  ⟨increase single operand reference counter 28.1⟩
}

mtdag_node(mtdag_node* a,const int dd,const constructortype con)
:I(root(a->I,dd)),node_data(0),X(a),Y(0),count(1),node_type(make_nt(dd,con)){
  assert(contype() == ROOT);
  assert(degree() == dd);
  assert(degree() >= 2);

  ⟨increase single operand reference counter 28.1⟩
}
```

⟨*increase operands reference counter* 27.2⟩≡ (27.1)

```
assert(X!=0); X->ref_plus();
assert(Y!=0); Y->ref_plus();
```

⟨*increase single operand reference counter* 28.1⟩≡ (27.1)
```
assert(X!=0); X->ref_plus();
```

⟨*destructor* 28.2⟩≡ (21)
```
~mtdag_node(){
  delete node_data;

  if (X && (X->ref_minus() == 0)) delete X;
  if (Y && (Y->ref_minus() == 0)) delete Y;
}
```

API methods

⟨*guaranteeing error* 28.3⟩≡ (21)
⟨*guaranteeing relative and absolute error* 28.4⟩
⟨*accessing internal approximation and error* 28.6⟩

⟨*guaranteeing relative and absolute error* 28.4⟩≡ (28.3) 28.5▷
```
void guarantee_absolute_error_two_to(const Exponent& p){
  init_node_data();
  guarantee_bound_two_to(p);
}
```

⟨*guaranteeing relative and absolute error* 28.4⟩+≡ (28.3) ◁28.4
```
void guarantee_relative_error_two_to(const Exponent& p){
  if(sign_with_separation_bound()==0) return;
  init_node_data();
  if(exact()) return;

  Approximation tmp;
  AP::abs(tmp,get_approx(),AP::error_prec(),AP::round_down());
  AP::sub(tmp,tmp,get_error(),AP::error_prec(),AP::round_down());
  assert(AP::sign(tmp) > 0);

  const Exponent floor_log2_z_low(AP::floor_log2(tmp));
  const Exponent q(p + floor_log2_z_low);

  guarantee_bound_two_to(q);
}
```

⟨*accessing internal approximation and error* 28.6⟩≡ (28.3) 29.1▷
```
void get_approx_and_error(Approximation& a, Approximation& e){
  init_node_data();
  a = get_approx();
  e = get_error();
}
```

⟨*accessing internal approximation and error* 28.6⟩+≡ (28.3) ◁28.6

```
const FilterPolicy& get_interval(){
  if(contype()==DOUBLE) return I;
  init_node_data();
  adjust_interval();
  return I;
}
```

Separation bound computation

⟨*separation bound concrete computation* 29.2⟩≡ (13.3)

```
void init_sep_bd(){

  switch(contype()) {
    case DOUBLE:
      assert(I.is_singleton());
      sep_bd().set(I.get_point());
      break;
    case BIGFLOAT:
      sep_bd().set(get_approx());
      break;
    case NEGATION:
      sep_bd().negation(X->sep_bd());
      break;
    case ROOT:
      sep_bd().root(X->sep_bd(),degree());
      break;
    case ADDITION:
      sep_bd().addition(X->sep_bd(),Y->sep_bd());
      break;
    case SUBTRACTION:
      sep_bd().subtraction(X->sep_bd(),Y->sep_bd());
      break;
    case MULTIPLICATION:
      sep_bd().multiplication(X->sep_bd(),Y->sep_bd());
      break;
    case DIVISION:
      sep_bd().division(X->sep_bd(),Y->sep_bd());
      break;
  }
}
```

⟨*sign computation* 29.3⟩≡ (21)

⟨*sign computation wrapper* 30.1⟩

⟨*sign computation auxillary methods* 30.2⟩

⟨*sign computation wrapper* 30.1⟩≡ (29.3)

```
int sign_with_separation_bound(){
  if(!I.contains(0.0)){
    return I.get_point() > 0.0 ? 1 : -1;
  }else if(I.is_singleton()){
    return 0;
  }

  init_node_data();
  separate_from_zero();

  return sign();
}
```

⟨*sign computation auxillary methods* 30.2⟩≡ (29.3)

```
inline void separate_from_zero() {
  if (!exact() && (is_zero() || floor_log2_approx() <= ceil_log2_error())) {
    Exponent log2_abserr = ceil_log2_error();
    Exponent log2_relerr = -27;

    do {
      log2_relerr *= 2;
      log2_abserr += log2_relerr;

      guarantee_bound_two_to(log2_abserr);
    } while(!exact() &&
            (is_zero() || floor_log2_approx() <= ceil_log2_error()));
  }
}
```

Fixed precision computation

⟨*fixed precision computation* 30.3⟩≡ (21)

⟨*fixed precision wrapper* 30.4⟩

⟨*fixed precision routines* 31.2⟩

⟨*fixed precision wrapper* 30.4⟩≡ (30.3)

```
void compute_op(const Precision p){

  switch(contype()){
    case DOUBLE:
      assert(I.is_singleton());
      approx() = Approximation(I.get_point());
      assert(exact());
      return;
    case BIGFLOAT:
      assert(exact());
      return;
    case NEGATION:        compute_neg();    break;
    case ROOT:            compute_root(p); break;
    case ADDITION:        compute_add(p);  break;
    case SUBTRACTION:     compute_sub(p);  break;
    case MULTIPLICATION:  compute_mul(p);  break;
```

```
    case DIVISION:        compute_div(p);  break;
  }

  if(exact()){
    convert_to_bigfloat();
  }
  else {
    requested_error() = ceil_log2_error();
  }

  ⟨try to improve filter policy 31.1⟩
}
```

⟨try to improve filter policy 31.1⟩≡ (13.4 30.4)

```
if(I.can_be_improved()) adjust_interval();
```

⟨fixed precision routines 31.2⟩≡ (30.3)

```
inline void compute_neg() {
  AP::neg(approx(),X->get_approx(),AP::get_prec(X->get_approx()),AP::round_to_nearest());
  error() = X->get_error();
}

void compute_add(const Precision p){
  assert(contype() == ADDITION);
  const bool isexact =
  AP::add(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

  AP::add(error(),X->get_error(),Y->get_error(),AP::error_prec(),AP::round_up());
  Approximation tmp;
  ⟨compute and add operation error 33.1⟩
}

void compute_sub(const Precision p){
  assert(contype() == SUBTRACTION);
  const bool isexact =
  AP::sub(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

  AP::add(error(),X->get_error(),Y->get_error(),AP::error_prec(),AP::round_up());
  Approximation tmp;
  ⟨compute and add operation error 33.1⟩
}

void compute_mul(const Precision p){
  assert(contype() == MULTIPLICATION);
  const bool isexact =
  AP::mul(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

  Approximation tmp;
  AP::abs(tmp,X->get_approx(),AP::error_prec(),AP::round_up());
  AP::mul(error(),tmp,Y->get_error(),AP::error_prec(),AP::round_up());

  AP::abs(tmp,Y->get_approx(),AP::error_prec(),AP::round_up());
  AP::add(tmp,tmp,Y->get_error(),AP::error_prec(),AP::round_up());
  AP::mul(tmp,tmp,X->get_error(),AP::error_prec(),AP::round_up());
```

```
  AP::add(error(),get_error(),tmp,AP::error_prec(),AP::round_up());
  ⟨compute and add operation error 33.1⟩
}

void compute_div(const Precision p){
  assert(contype() == DIVISION);

  if(Y->sign_with_separation_bound() == 0)
    real_algebraic_error_handler(1,__FILE__,__LINE__,RA_FUNCTION,
                                 "Division by zero.");

  const bool isexact =
  AP::div(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

  Approximation tmp;
  AP::abs(tmp,Y->get_approx(),AP::error_prec(),AP::round_down());
  AP::sub(tmp,tmp,Y->get_error(),AP::error_prec(),AP::round_down());
  assert(AP::sign(tmp) > 0);
  Exponent minus_floor_log2_ylow = -AP::floor_log2(tmp);
}

void compute_root(const Precision p){
  assert(contype() == ROOT);

  if(X->sign_with_separation_bound() < 0)
    real_algebraic_error_handler(1,__FILE__,__LINE__,RA_FUNCTION,
                                 "Root of a negative number.");

  const bool isexact =
  AP::root(approx(),X->get_approx(),degree(),p,AP::round_to_nearest());

  if(X->is_zero()){
    assert(X->exact());
    AP::zeroize(error());
  } else {
    Approximation tmp;
    AP::sub(tmp,X->get_approx(),X->get_error(),AP::error_prec(),AP::round_down());
    assert(AP::sign(tmp) > 0);

    const int d = degree();
    int one_plus_floor_log2_d;
    std::frexp(static_cast<double>(d),&one_plus_floor_log2_d);

    Exponent log2_error_term =
       (static_cast<Exponent>(1-d) * AP::floor_log2(tmp)) / static_cast<Exponent>(d)
      + static_cast<Exponent>( 2 - one_plus_floor_log2_d) ;
    AP::ldexp(error(),X->get_error(),log2_error_term);
    ⟨compute and add operation error 33.1⟩
  }
}
```

⟨*compute and add operation error* 33.1⟩≡ (31.2)

```
if(!isexact){
  AP::abs(tmp,get_approx(),AP::error_prec(),AP::round_up());
  AP::ldexp(tmp,tmp,-static_cast<Exponent>(p));
  AP::add(error(),get_error(),tmp,AP::error_prec(),AP::round_up());
}
```

Accuracy-driven computation overview

⟨*accuracy driven computation* 33.2⟩≡ (21)

⟨*adc wrapper* 14.2⟩
⟨*adc error propagation* 33.5⟩
⟨*adc recomputation* 35⟩
⟨*adc fixup* 13.4⟩
⟨*adc auxillary functions* 26.3⟩

Error propagation for accuracy-driven computation

⟨*propagate error topologically descending* 33.3⟩≡ (14.2)

```
for(std::vector<SelfType*>::const_reverse_iterator it=order.rbegin();it!=order.rend();++it)
{
  assert((*it)->contype() != DOUBLE);
  assert((*it)->contype() != BIGFLOAT);

  switch((*it)->contype()) {
    case NEGATION:       (*it)->prop_neg_error();  break;
    case ROOT:           (*it)->prop_root_error(); break;
    case ADDITION:
    case SUBTRACTION:    (*it)->prop_add_or_sub_error();  break;
    case MULTIPLICATION: (*it)->prop_mul_error();  break;
    case DIVISION:       (*it)->prop_div_error();  break;
  }
}
```

⟨*set q and check exactness* 33.4⟩≡ (33.5 35 36.2)

```
assert(!exact());

const Exponent q = get_requested_error();
if(ceil_log2_error() <= q) return;
```

⟨*adc error propagation* 33.5⟩≡ (33.2)

```
void prop_neg_error(){
  assert(contype() == NEGATION);

  ⟨set q and check exactness 33.4⟩

  if(q < X->get_requested_error()) X->requested_error() = q;
}

void prop_root_error(){
  assert(contype() == ROOT);
```

```
  ⟨set q and check exactness 33.4⟩

  if( !X->is_zero() ){
    const Exponent floor_log2_xlow = X->floor_log2_low();

    const int d = degree();
    int one_plus_floor_log2_d;
    std::frexp(static_cast<double>(d),&one_plus_floor_log2_d);

    const Exponent qx =
    std::min(floor_log2_xlow - static_cast<Exponent>(1),
            (static_cast<Exponent>(d-1) * floor_log2_xlow) / static_cast<Exponent>(d)
             + q + static_cast<Exponent>(one_plus_floor_log2_d - 3));

    if(qx < X->get_requested_error()) X->requested_error() = qx;
  }
}

void prop_add_or_sub_error(){
  assert(contype() == ADDITION || contype() == SUBTRACTION);

  ⟨set q and check exactness 33.4⟩

  const Exponent qx = q - static_cast<Exponent>(2);
  if(qx < X->get_requested_error()) X->requested_error() = qx;
  if(qx < Y->get_requested_error()) Y->requested_error() = qx;
}

void prop_mul_error(){
  assert(contype() == MULTIPLICATION);

  ⟨set q and check exactness 33.4⟩

  if ((X->is_zero() && X->exact()) ||
      (Y->is_zero() && Y->exact())) return;

  const Exponent two(2);
  const Exponent c = q - two;

  Exponent qx = c - Y->ceil_log2_high();
  Exponent qy = c - X->ceil_log2_high();

  if(qx + qy > c){
    qx = (c+qx-qy)/two;
    qy = c - qx;
  }

  assert(qx <= q - static_cast<Exponent>(2) - Y->ceil_log2_high());
  assert(qy <= q - static_cast<Exponent>(2) - X->ceil_log2_high());
  assert(qx+qy <= q - static_cast<Exponent>(2));

  if(qx < X->get_requested_error()) X->requested_error() = qx;
  if(qy < Y->get_requested_error()) Y->requested_error() = qy;
}

void prop_div_error(){
```

```
  assert(contype() == DIVISION);

  ⟨set q and check exactness 33.4⟩

  if(X->is_zero() && X->exact()) return;

  const Exponent floor_log2_ylow = Y->floor_log2_low();

  const Exponent qx = q - static_cast<Exponent>(4) + floor_log2_ylow;
  const Exponent qy = std::min(floor_log2_ylow - static_cast<Exponent>(1),
                               q - static_cast<Exponent>(4) - X->ceil_log2_high()
                                  + static_cast<Exponent>(2)*floor_log2_ylow);

  if(qx < X->get_requested_error()) X->requested_error() = qx;
  if(qy < Y->get_requested_error()) Y->requested_error() = qy;
}
```

Recomputation methods for accuracy-driven computation

⟨*adc recomputation* 35⟩≡ (33.2) 36.2▷

```
void recompute_neg(){
  assert(contype() == NEGATION);

  ⟨set q and check exactness 33.4⟩

  AP::neg(approx(),X->get_approx(),AP::get_prec(X->get_approx()),AP::round_to_nearest());
  error() = X->get_error();
}

void recompute_root(){
  assert(contype() == ROOT);

  ⟨set q and check exactness 33.4⟩

  Precision p(AP::min_prec());
  int d = degree();

  if( !X->is_zero() ){
    const Exponent relerr = X->ceil_log2_approx()/static_cast<Exponent>(d)
                                                  + static_cast<Exponent>(2) - q;
    p = AP::convert_to_prec(relerr);
  }

  const bool isexact = AP::root(approx(),X->get_approx(),d,p,AP::round_to_nearest());

  ⟨set new error for unary operators 37.1⟩
}

void recompute_add(){
  assert(contype() == ADDITION);

  ⟨set q and check exactness 33.4⟩
  ⟨set p for addition and subtraction 36.1⟩

  const bool isexact =
```

```
    AP::add(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

    ⟨set new error for binary operators 37.2⟩
  }

  void recompute_sub(){
    assert(contype() == SUBTRACTION);

    ⟨set q and check exactness 33.4⟩
    ⟨set p for addition and subtraction 36.1⟩

    const bool isexact =
    AP::sub(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());

    ⟨set new error for binary operators 37.2⟩
  }
```

⟨set p for addition and subtraction 36.1⟩≡ (35)

```
  Precision p(AP::min_prec());
  if (!X->is_zero() && !Y->is_zero()) {
    const Exponent relerr = std::max(X->ceil_log2_approx(),
                                     Y->ceil_log2_approx())
                                                  + static_cast<Exponent>(2) - q;
    p = AP::convert_to_prec(relerr);
  } else if (!X->is_zero()) {
    const Exponent relerr = X->ceil_log2_approx() + static_cast<Exponent>(1) - q;
    p = AP::convert_to_prec(relerr);
  } else if (!Y->is_zero()) {
    const Exponent relerr = Y->ceil_log2_approx() + static_cast<Exponent>(1) - q;
    p = AP::convert_to_prec(relerr);
  }
```

⟨adc recomputation 35⟩+≡ (33.2) ◁35

```
  void recompute_mul(){
    assert(contype() == MULTIPLICATION);

    ⟨set q and check exactness 33.4⟩

    bool isexact = false;
    if( !X->is_zero() && !Y->is_zero() ){
      const Exponent two(2);
      const Exponent relerr = X->ceil_log2_approx()
                        + Y->ceil_log2_approx() + two - q;

      const Precision p = AP::convert_to_prec(relerr);

      isexact =
      AP::mul(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());
    }
    else
    {
      isexact = true;
      AP::zeroize(approx());
    }
```

```
    ⟨set new error for binary operators 37.2⟩
  }

  void recompute_div(){
    assert(contype() == DIVISION);

    ⟨set q and check exactness 33.4⟩

    assert(!Y->is_zero());

    bool isexact = false;
    if( !X->is_zero() ){
      const Exponent relerr = X->ceil_log2_approx()
                            - Y->floor_log2_approx() + static_cast<Exponent>(1) - q;
      const Precision p = AP::convert_to_prec(relerr);

      isexact =
      AP::div(approx(),X->get_approx(),Y->get_approx(),p,AP::round_to_nearest());
    }
    else
    {
      AP::zeroize(approx());
      isexact = true;
    }

    ⟨set new error for binary operators 37.2⟩
  }
```

⟨*set new error for unary operators* 37.1⟩≡ (35)

```
  if ( isexact && X->exact() ) AP::zeroize(error());
  else AP::pow2(error(),q);
```

⟨*set new error for binary operators* 37.2⟩≡ (35 36.2)

```
  if ( isexact && X->exact() && Y->exact() ) AP::zeroize(error());
  else AP::pow2(error(),q);
```