COCONUT Deliverable D13

Upgraded State of the Art Techniques Implemented as Modules

July 29, 2003

ILOG
TU Darmstadt
University of Coimbra
University of Vienna
University of Nantes
Catholic University of Louvain
Swiss Federal Institute of Technology

CHAPTER 5

UWien Evaluators

H. SCHICHL
University of Vienna, Department of Mathematics
A-1090 Wien, Austria
email: Hermann.Schichl@esi.ac.at

1. Introduction

Evaluators are small modules in the API which can be used by inference engines, and which provide information about certain aspects of the model DAG or parts of the model DAG.

The evaluators were designed as intermediate layer between the inference engines and the internal representation of optimization problems. Their outside structure more or less independent of DAGs, so if the internal representation would change in the future, inference engines using evaluators should not need lots of updates.

All evaluators are constructed from a very small number of base classes. They can make use of the `variable_indicator`, a bitmap recording which variables every node depends on.

For a full class description see the reference manual of the COCONUT API Version 2.

2. Base class

There are four different base classes for evaluator programming, depending on the type of evaluation:

- forward evaluation,
- backward evaluation,
- forward evaluation with caching,
- backward evaluation with caching.

For an evaluator, a number of methods have to be provided:

`initialize`: This method is called for every node before children are visited. For forward evaluation the method should return an `int`. If the return value is negative, the `short_cut_to` method will be called for performing a short-cut. If the return value is $i > 0$, then skip $i - 1$ children in the evaluation. If the return value is zero, then do not walk to the children.

`calculate`: For forward evaluation this method is called after all children have been visited, for backward evaluation it is called right before the first child is visited. For backward evaluation a return value of type `int` is expected, which follows the same rule as the return value of `initialize` for forward evaluators.

`calculate value`: This method is called before the evaluator is destroyed. It should return the evaluator's return value.

`cleanup`: This method is called just before `calculate_value`, and it should be used to get rid of allocated data,...

`update`: The `update` method is called every time after a child has finished. The second argument is the return value of the child. The method should return `int`. This number is the number of children which should be skipped during evaluation. If the return value is negative, no more children are visited.

`short_cut_to`: This method is only called, if initialize (for forward) or calculate (for backward) return a negative number. It should return a walker pointing to the point the short-cut leads to. The graph walk is continued there.

`is_cached`: This method should return a `bool` specifying whether the result of this node is already in the cache.

retrieve_from_cache: If `is_cached` returns `true`, this method is called. It should retrieve from the cache the result and store it somewhere such that `calculate_value` can produce a proper return value.

All evaluators are used in the same way. First they are constructed using all the relevant data. Then the `evaluate` function is called. This function takes two arguments. The first is the evaluator, the second argument is a `walker` of the model DAG pointing to the node (e.g., objective, constraint) which should be evaluated. The evaluate function returns the return value the evaluator provides for the start node.

3. Implemented Evaluators

A number of evaluators have been implemented in the COCONUT API version 2. In the following I will give a short description of all of them.

3.1. Function Evaluation. This is the simplest evaluator. It provides a real function evaluation. The constructor has the form

```
func_eval(const std::vector<double>& __x, const variable_indicator& __v,
          const model& __m, std::vector<double>* __c) : _Base()
```

Here `__x` defines the point on which the function is evaluated. `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

Example: In this code snipplet the objective function is evaluated.

```
#include <model.h>
#include <func_evaluator.h>

int main()
{
```

```
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  func_eval fv(x, v_ind, model, NULL);
  double fx = evaluate(fv, DAG.objective);
  std::cout << "The function value is: " << fx << std::endl;
  return 0;
}
```

3.2. Gradient Evaluation. Gradient evaluation consists of **three** evaluators: `prep_d_eval`, `func_d_eval`, and `der_eval`.

The evaluator utilizes the backward mode of automatic differentiation. For this the partial derivatives of every node w.r.t. its children have to be computed. This is done in forward mode during a function evaluation.

The `prep_d_eval` evaluator is used to set up the data structure for automatic differentiation. It has to be called before `func_d_eval` is called. Its constructor is

```
prep_d_eval(std::vector<std::vector<double> >& __d,
            unsigned int _num_of_nodes)
```

where `__d` is a reference to the automatic differentiation data structure.

After `prep_d_eval` has set up the data structure, `func_d_eval` is used to calculate a function value, like with `func_eval` with the difference that all partial derivatives in the tree are kept within the already set up data structure. The constructor of `func_d_eval` is:

```
func_d_eval(const std::vector<double>& __x, const variable_indicator& __v,
            const model& __m, std::vector<std::vector<double> >& __d,
            std::vector<double>* __c)
```

where `__x`, `__v`, `__m`, and `__c` have the same semantics as in `func_eval` (see Section 3.1). The parameter `__d` is a reference to the automatic differentiation data structure.

After the function evaluation the third evaluator `der_eval` is called, which computes the gradient. The constructor is

```
der_eval(std::vector<std::vector<double> >& __der_data,
         variable_indicator& __v, const model& __m,
         std::vector<std::vector<double > >* __d,
```

```
            std::vector<double>& __grad)
```

Here __der_data is a reference to the derivative structure filled by func_d_eval before. There is no internal status or storage in __der_eval, and all information is passed via __der_data. The parameters __v and __m are as before, and __d is the cache for derivative evaluation. Finally, the parameter __grad is a reference to a vector, which has to be initialized by 0, into which the gradient is stored.

Example: The following piece of code evaluated the gradient of the objective function.

```
#include <model.h>
#include <der_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> x(n), grad(n), zero(n, 0.);
  std::vector<std::vector<double> > d_data;
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  //prepare data structure for derivative
  prep_d_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_d_eval fv(x, v_ind, model, d_data, NULL);
  double fx = evaluate(fv, DAG.objective);

  der_eval dv(d_data, v_ind, DAG, NULL, grad);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function value is: " << fx << std::endl;
  std::cout << "The gradient is: ";
  std::copy(grad.begin(), grad.end(),
            std::ostream_iterator<double>(std::cout, ", "));
  return 0;
```

```
}
```

3.3. Range Evaluation (Interval Evaluation). This is the simplest interval evaluator. It provides a function range enclosure using interval arithmetic. The constructor has the form

```
interval_eval(const std::vector<interval>& __x, const variable_indicator& __v,
              const model& __m, std::vector<interval>* __c) : _Base()
```

Here `__x` defines the box over which the function range is enclosed. `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

Example: In this code snipplet the range of the objective function is enclosed.

```
#include <model.h>
#include <int_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  interval_eval fv(x, v_ind, model, NULL);
  interval fx = evaluate(fv, DAG.objective);
  std::cout << "The function range is in: " << fx << std::endl;
  return 0;
}
```

3.4. Interval Derivative Evaluation. This evaluator set provides range enclosures for gradients. It consists of **three** evaluators, like gradient evaluation (see Section 3.2): `prep_id_eval`, `func_id_eval`, and `ider_eval`.

The evaluator utilizes the backward mode of automatic differentiation. For this the partial derivatives of every node w.r.t. its children have to be computed. This is done in forward mode during a function range enclosure. The special results on improved ranges due to exploiting the results of constraint propagation by [**8**] have been implemented.

The `prep_id_eval` evaluator is used to set up the data structure for automatic differentiation. It has to be called before `func_id_eval` is called. Its constructor is

```
prep_id_eval(std::vector<std::vector<interval> >& __d,
             unsigned int _num_of_nodes)
```

where `__d` is a reference to the automatic differentiation data structure.

After `prep_id_eval` has set up the data structure, `func_id_eval` is used to calculate a function range enclosure, like with `interval_eval` with the difference that ranges of partial derivatives in the tree are kept within the already set up data structure. The constructor of `func_id_eval` is:

```
func_id_eval(const std::vector<interval>& __x, const std::vector<interval> __rg,
             const variable_indicator& __v, const model& __m,
             std::vector<std::vector<interval> >& __d,
             std::vector<interval>* __c)
```

where `__x`, `__v`, `__m`, and `__c` have the same semantics as in `interval_eval` (see Section 3.3). The parameter `__d` is a reference to the automatic differentiation data structure. The `__rg` vector contains the best known range of all nodes in the DAG (even the intermediate ones).

After the range evaluation the third evaluator `ider_eval` is called, which computes the gradient. The constructor is

```
ider_eval(std::vector<interval> __x,
          std::vector<std::vector<interval> >& __ider_data,
          variable_indicator& __v, const model& __m,
          std::vector<std::vector<interval> >* __d,
          std::vector<interval>& __grad)
```

Here `__ider_data` is a reference to the derivative structure filled by `func_id_eval` before. There is no internal status or storage in `__ider_eval`, and all information is passed via `__ider_data`. The parameters `__v` and `__m` are as before, and `__d` is the cache for derivative evaluation. Finally, the parameter `__grad` is a reference to a vector, which has to be initialized by 0, into which the gradient is stored. The first parameter `__x` is a reference to the box over which the gradient should be enclosed.

Example: The following code piece evaluates the range of the gradient of the objective function.

```
#include <model.h>
#include <ider_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<interval> x(n), grad(n), zero(n, 0.);
  std::vector<interval> rg(DAG.number_of_nodes());
  std::vector<std::vector<interval> > d_data;
  variable_indicator v_ind(n);
```

```
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  /**
   * constraint propagation happens here,
   * storing the node ranges in rg.
   */

  //prepare data structure for derivative
  prep_id_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_id_eval fv(x, rg, v_ind, model, d_data, NULL);
  interval fx = evaluate(fv, DAG.objective);

  der_eval dv(x, d_data, v_ind, DAG, NULL, grad);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function range is: " << fx << std::endl;
  std::cout << "The interval derivative is: ";
  std::copy(grad.begin(), grad.end(),
            std::ostream_iterator<interval>(std::cout, ", "));
  return 0;
}
```

3.5. First order Slope Evaluation. This evaluator set provides range first order slopes for functions. It consists of **three** evaluators, like gradient evaluation (see Section 3.2): `prep_islp_eval`, `func_islp_eval`, and `islp_eval`.

The evaluator utilizes the backward mode of automatic differentiation. For this the partial slopes of every node w.r.t. its children have to be computed. This is done in forward mode during a function evaluation on the center of the slope. The special results on improved ranges due to exploiting the results of constraint propagation by [**8**] have been implemented.

The `prep_islp_eval` evaluator is used to set up the data structure for automatic differentiation. It has to be called before `func_islp_eval` is called. Its constructor is

```
  prep_id_eval(std::vector<std::vector<interval> >& __d,
               unsigned int _num_of_nodes)
```

where `__d` is a reference to the automatic differentiation data structure.

After `prep_islp_eval` has set up the data structure, `func_islp_eval` is used to calculate a function range enclosure, like with `interval_eval` with the difference that ranges of partial slopes in the tree are kept within the already set up data structure. The constructor of `func_islp_eval` is:

```
func_islp_eval(const std::vector<double>& __z, const std::vector<interval> __rg,
               const variable_indicator& __v, const model& __m,
               std::vector<std::vector<interval> >& __d,
               std::vector<double>& __f)
```

where `__z` is the center of the slope, and `__v`, and `__m` have the same semantics as in `interval_eval` (see Section 3.3). The parameter `__d` is a reference to the automatic differentiation data structure. The `__rg` vector contains the best known range of all nodes in the DAG (even the intermediate ones). The parameter `__f` is used to store center information. The evaluator returns a value of type `func_islp_return_type` consisting of an approximate function evaluation at the center z, a function range enclosure over the point interval $\mathbf{z} = [z, z]$, and a function range enclosure over the box, over which the slopes are to be calculated.

After the function evaluation at the center the third evaluator `islp_eval` is called, which computes the slope. The constructor is

```
islp_eval(std::vector<std::vector<interval> >& __islp_data,
          variable_indicator& __v, const model& __m,
          std::vector<std::vector<interval> >* __d,
          std::vector<interval>& __islp)
```

Here `__islp_data` is a reference to the derivative structure filled by `func_id_eval` before. There is no internal status or storage in `__islp_eval`, and all information is passed via `__islp_data`. The parameters `__v` and `__m` are as before, and `__d` is the cache for slope evaluation. Finally, the parameter `__islp` is a reference to a vector, which has to be initialized by 0, into which the slope is stored.

Example: The following code piece evaluates the slope of the objective function.

```
#include <model.h>
#include <islp_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<double> z(n), f(DAG.number_of_nodes());
  std::vector<interval> x(n), slope(n), zero(n, 0.);
  std::vector<interval> rg(DAG.number_of_nodes());
  std::vector<std::vector<interval> > slp_data;
  variable_indicator v_ind(n);
  v_ind.set(0,n);
```

```
  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }

  /**
   * constraint propagation happens here,
   * storing the node ranges in rg.
   */

  //prepare data structure for derivative
  prep_islp_eval pd(d_data, DAG.number_of_nodes());
  evaluate(pd, DAG.ground());

  //initialize variables for evaluation
  func_islp_eval fv(z, rg, v_ind, model, d_data, f);
  interval fx = evaluate(fv, DAG.objective);

  der_eval dv(x, d_data, v_ind, DAG, NULL, slope);
  grad = zero;
  evaluate(dv, DAG.objective);

  std::cout << "The function range is: " << fx.rg << std::endl;
  std::cout << "The enclosure of the center evaluation is: " << fx.fi << std::endl;
  std::cout << "The approximation of the center evaluation is: " << fx.f << std::endl;
  std::cout << "The slope is: ";
  std::copy(slope.begin(), slope.end(),
            std::ostream_iterator<interval>(std::cout, ", "));
  return 0;
}
```

3.6. Analytic-Differentiable Evaluation. This evaluator produces in one sweep

- an interval for range evaluation
- an interval for derivative evaluation
- a complex (rectangular) interval symmetric with respect to the real axis (consisting of an interval representing the real part and a double representing the maximal distance from the real axis).

The constructor has the form

```
analyticd_eval(const std::vector<analyticd>& __x, const variable_indicator& __v,
               const model& __m, std::vector<analyticd>* __c) : _Base()
```

Here `__x` defines the box over which the evaluation is done, and `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

This evaluator is not yet implemented for many operators, but the most important ones are covered.

Example: The following code snipplet evaluates the objective function.

```
#include <model.h>
#include <ade_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<analyticd> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  /**
   *  Here initialize x
   */

  analyticd_eval fv(x, v_ind, model, NULL);
  analyticd fx = evaluate(fv, DAG.objective);
  return 0;
}
```

3.7. Bounded Interval Evaluation. This is an interval evaluator for bounded evaluation. It provides a function range enclosure using interval arithmetic avoiding infinities. The constructor has the form

```
b_interval_eval(const std::vector<b_interval>& __x, const variable_indicator& __v,
               const model& __m, std::vector<b_interval>* __c) : _Base()
```

Here `__x` defines the box over which the function range is enclosed. `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

Example: This code snipplet encloses the objective function:

```
#include <model.h>
#include <bint_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
```

```
  std::vector<b_interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  b_interval_eval fv(x, v_ind, model, NULL);
  b_interval fx = evaluate(fv, DAG.objective);
  // produce output
  return 0;
}
```

3.8. Complex Interval Evaluation. This evaluator provides enclosures of the function in complex (rectangular) intervals symmetric w.r.t. the real axis. The constructor has the form

```
c_interval_eval(const std::vector<c_interval>& __x, const variable_indicator& __v,
                const model& __m, std::vector<c_interval>* __c) : _Base()
```

Here `__x` defines the box over which the function range is enclosed. `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

Example: In this code snipplet the objective function is evaluated.

```
#include <model.h>
#include <cint_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<c_interval> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  c_interval_eval fv(x, v_ind, model, NULL);
  c_interval fx = evaluate(fv, DAG.objective);
```

```
  // produce output
  return 0;
}
```

3.9. Infinity-Bound Evaluation. This is an evaluator doing asymptotic arithmetic. It provides a function enclosure at $\pm\infty$. The constructor has the form

```
infbound_eval(const std::vector<infbound>& __x, const variable_indicator& __v,
              const model& __m, std::vector<infbound>* __c) : _Base()
```

Here `__x` defines the asymptotic region over which the function is enclosed. `__v` is a variable indicator, where all variables are `set`, if they have been changed since the last evaluation. The parameter `__m` contains a reference to the actual model, and `__c` is the cache.

Example: In this code the objective function is enclosed.

```
#include <model.h>
#include <infb_evaluator.h>

int main()
{
  model DAG("t.dag");
  int n = DAG.number_of_variables();
  std::vector<infbound> x(n);
  variable_indicator v_ind(n);
  v_ind.set(0,n);

  std::cout << "x(1:" << n << ") = ";
  // read evaluation point
  for(i=0; i<n; i++)
  {
    std::cin >> x[i];
  }
  infbound_eval fv(x, v_ind, model, NULL);
  infbound fx = evaluate(fv, DAG.objective);
  // output
  return 0;
}
```