The COCONUT Project

Deliverable D6

SPECIFICATION OF MODULES INTERFACE

INTERNAL REPRESENTATION AND MODULES API

Contract number : IST-2000-26063

Document type : Deliverable
Classification : Public
Version : 1.3
Due date : 1 September 2001

Status : final
Delivery date : 26 December 2001

Editor : Christian Bliek, ILOG
Contributors : Christian Bliek *ILOG.*
Hermann Schichl *UWIEN*

Contents

Module API

This document contains the first description of the application programmer's interface (API). Its definition is as specific as possible before the coding of inference engines has started. We expect the overall structure not to change much but some adjustments in API details will probably have to occur. Especially the research on the strategy engine is very likely to require additions to the API. This document should, however, contain all the basics to get started.

In section 2 the structure of inference engine modules is described. The structure of the search graph, including the incrementality concept by deltas is described next in section 3. The internal representation of the optimization problem, the model, is defined as a collection of a number of classes in section 4. At the end in appendix A a description of the graph template library (GTL) is given, which is, at least to start with, used to implement the directed acyclic graphs (DAG) used as a basic structure in some places in this API.

1 Introduction

The API we describe in this document is a basic one. It corresponds with an initial simple architecture. This architecture is illustrated in figure 1. It consists

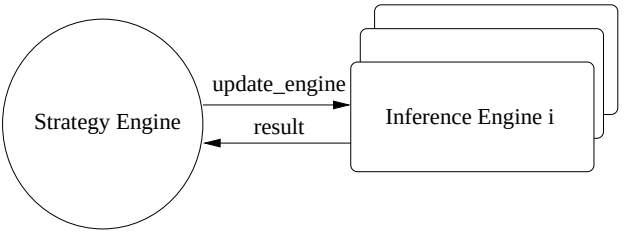

Figure 1: Initial Architecture

of various inference modules and a strategy engine. The inference modules are able to make inferences based on a nonlinear model. Examples of inference modules are a local nonlinear solver, a linear or a quadratic programming solver, a constraint propagation engine or an interval arithmetic engine. Inside the

strategy engine the modules are called so as to find a solution to the nonlinear problem. The API described in this document will allow the execution of basic strategies. For example through this API it will be possible to program the exploration of a search tree and call various inference modules at each node.

The basic architecture provides a common control framework for combining techniques. In the initial implementation, strategies will be hard-coded for exploration and testing purposes (task T320). We envision that in the future, the strategy engine will be programmed using a high level language and that it will have at its disposal various management submodules to manipulate models and modules automatically: this will be developed as part of task T330 of the work plan. The API proposed in this document can be extended to accommodate for more complicated architectures of this type in the future.

We now give a brief overview of the initial architecture we will implement. The strategy engine will use an explicit representation of the search tree. As illustrated in figure 2 a search tree is a DAG of search nodes. Each search node

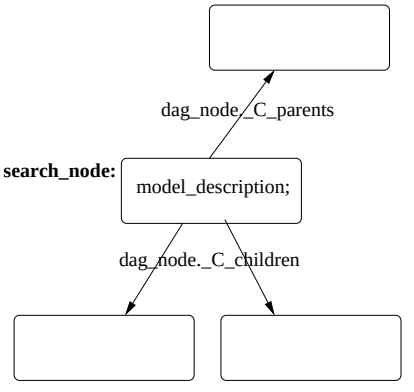

Figure 2: Search Trees are a DAGs of Search Nodes

contains a description of a model. This description can be a pointer to a model, or an incremental modification with respect to another model. The parent and child nodes are available via the DAG.

Nonlinear problems are represented as models, and models are represented by DAGs of expression nodes (see figure 3). The expression nodes correspond to arithmetic operators; the edges represent dependencies. An expression node is described by the type of the arithmetic operator (stored in the field `expression_type`) and the constraints imposed on the expression nodes (represented by the bounds stored in the field `f_bounds`). The parents and children of an expression node can be found via the DAG. When exploring a search tree the strategy engine will need to notify the inference modules on the choice points it makes. This can be done under the form of incremental modifications to the current model.

The remainder of this document describes the API of the various classes needed in this architecture.

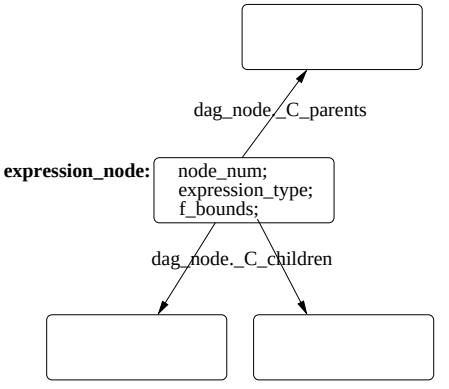

Figure 3: Models are DAGs of Expression Nodes

2 Inference Engines

This section contains the base class of all inference engines. Every engine written will be a subclass of this base class.

```
class inference_engine
{
public:
  // the constructor for a basic inference engine
  inference_engine(const search_node& search_node);
  inference_engine();
  ~inference_engine();

  virtual bool update_engine(const model_delta& model_delta);
  virtual bool update_engine(const model& model);

  virtual const error_return& infer();

  // SOLUTION 1
  // the call providing the result is defined in the subclass
  // the strategy engine has info of what method
  // it should call for which subclass and what
  // the result type is
  // SOLUTION 2
  virtual model result();
  // SOLUTION 3
  virtual proposal result(string requested_result_type);

  const string& name() { return name; }
```

```
    // collect statistics
    virtual statistic_info last_call_stat();
    virtual statistic_info cumulative_stat();
};
```

There is a set of constructors for this class. Usually, the first one will be taken. It initializes the engine with a model representation in form of a search node. Later an existing engine can be updated by a model delta or a new full model using the `update_engine` call. The return value of this function signals whether the update was successful. If the inference engine does not support incremental changes, it must still have a function for incremental update, which should return `false` in that case.

Calling the `infer` method tells the inference engine to work on the model and prepare the result. It returns an `error_return` class, which will be specified more exactly later. For the beginning, lets set

```
typedef int error_return;
```

There are three possible solutions to pass return values from inference engines back to the strategy engine, which in the first version will be a simple hard coded C++ function.

1. Every engine could specify its own `result` function, we would use a document on the BSCW server to agree on a certain number of different return value types which could be used (this has to be done in order to e.g. prevent two different local solvers to return their results in different formats).

2. All engines could return a `model` or a `search_node` class, where every change is coded as DAG or as annotation.

3. There could be a class `proposal`, such that every engine returns a subclass of `proposal`, like

```
class proposal
{
public:
  vector<unsigned short> _weight; // importance of the results
  vector<certificate> _c;         // for debugging purposes and
                                  // verified computing (for later)
  // the subclass would add additional entries here and methods for
  // accessing these entries.
}
```

For the start we agreed to keep solution 1, since changing this later will not require much additional work.

The last three methods return the name of the inference engine (e.g. CPLEX) and provide statistical information collected for dynamic strategies and debugging. This is for later.

3 Search graphs

The models investigated are collected in a search graph. This graph is another DAG of `search_nodes`. This search graph is a representation of the search space and it will grow and shrink as new models are generated and some of them are solved.

The search node is defined by the following class

```
class search_node
{
public:
  bool uses_delta;  // delta vs. full description
  union
  {
    model       _model;
    model_delta _model_delta;
  }   model_description;

  annotation an;

  // constructors for each case and destructor
  search_node(const model& _model);
  search_node(const model_delta& _model_delta);
  ~search_node();
};
```

It distinguishes between an incremental change and a full model description. In addition it contains annotations containing best local points, Lagrange multipliers and the like.

While models are in more detail analyzed in section 3.1 we first take a look at the model deltas.

```
class model_delta
{
public:
  bound_deltas _constraint_deltas; // new bounds of expression_nodes

  model _new_constraints;          // dags that refer to ghostnodes

  vector<unsigned int> _rm_constraints; // expression_nodes that
                                        // need to be removed or
                                        // inactivated in the model
  // the subdag that has to be be removed if not inactivation is
  // chosen, it is stored in order to be able to undo this delta.
  // this might go away like old_f_bounds below.
  model _old_constraints;
};
```

```
  class bound_deltas
  {
  public:
    vector<unsigned int> indices;    // this is empty for updating all
                                     // variables
    vector<interval> new_f_bounds;   // new bounds
    vector<interval> old_f_bounds;   // original bounds for backtracking
                                     // might be removed.
  }
```

Such a delta consists of a bounds update and a model update. The bounds update can come in two flawors, a sparse and a full update. If there are no indices, all the nodes are to be updated. Otherwise the index vector tells, which bound concerns which expression node. There is the possibility to remove the `old_f_bounds`, because during backtracking, the bounds can be recalculated by an upwards walk of the search graph. It remains to investigate which choice is better.

The second component is the model update. It consists of a DAG of expression nodes whose leafs consist of *ghost nodes*. All the node numbers in this DAG have to be unique with respect to the full model DAG except for the ghost nodes. These are just empty nodes, for which only the node number matters. This node number specifies at which nodes in the model the new DAG has to be "glued". The constraints which are to be removed are just represented by the node number of their root nodes. They could be either *inactivated* using a flag or removed. If they are removed, they will be stored in DAG form in `_old_constraints`.

```
  class annotation
  {
  public:
    vector<double>* x_best;     // best point for this model
    vector<double>* L_mult;     // Lagrangian multipliers
    double f_best;              // best function value
    // ... others as needed

  public:
    annotation();
    ~annotation();
};
```

The `x_best` and `L_mult` are pointers to vectors. Thus, there is no need for an annotation delta, because the data structures are small enough to be copied as a whole. The annotation class will be filled during development, as it cannot easily be predicted now what information will be necessary during a complex solution process.

Eventually, the search graph is just a DAG of search nodes. Perhaps, we will later find the need to add additional construction methods and hence define a subclass instead of a simple `typedef`

```
typedef dag<search_node> search_graph;
```

3.1 Model

A model, represented by a DAG of expressions is a subclass of `dag<expression_node>` with a few additional methods to make construction of expression DAGs easier.

```
class model: public dag<expression_node>
{
private:
  unsigned int node_num_max;
  unsigned int num_of_vars;

public:
  expression_dag(int _num_of_var);

  int next_num();

  _ewalk constant(double _constant);
  _ewalk constant(vector<double> _constant);
  _ewalk variable(int _vnum);

  _ewalk binary(const _ewalk& _op1, const _ewalk& _op2, int expr_type);
  _ewalk binary(const _ewalk& _op1, const _ewalk& _op2, int expr_type,
                additional_info_u _meaning);

  _ewalk unary(const _ewalk& _op1, int expr_type);
  _ewalk unary(const _ewalk& _op1, int expr_type,
               additional_info_u _meaning);

  _ewalk nary(const vector<_ewalk>& _op, int expr_type);
  _ewalk nary(const vector<_ewalk>& _op, int expr_type,
              additional_info_u _meaning);
  _ewalk vnary(int expr_type, ...);
};
```

Here `_ewalk` is an expression walker (see section A.3 on walkers), which is a simple generalization of a pointer to an `expression_node` and can be used almost like a pointer.

`constant` creates a node for a constant, and `variable` one for a variable, `unary` creates a unary operation, `binary` a binary, and `nary` and `vnary` create n–ary (associative) operations.

4 Internal Representation

This section is devoted to the description of the internal representation of optimization problems in the COCONUT solver. In principle, every mathematical problem is represented as a directed acyclic graph (DAG), whose nodes correspond to arithmetic operations, variables, constants, or elementary functions. Some of them are builtin, others can be freely defined, as long as a set of C(++) functions is provided to compute all the necessary algorithms (evaluation, interval evaluation, derivative, ...).

A user of the optimization program will be able to enter general mathematical expressions and constraints in our system by using one of the supported modeling languages like AMPL, but internally the problem will be represented in a special form which is better suited for the solution process.

The internal representation of the program should mathematically look like follows: Every constraint is transformed into the form

$$F_i(x) = b_i^T x + x^T A_i x + s_i(x) + g_i(x) \in [F_i]$$

where $s_i(x)$ and $g_i(x)$ are not quadratic factorable, and $s_i(x)$ is separable. A constant in F_i can easily be absorbed into $[F_i]$. The vector b_i should be represented in sparse form. This makes it easy to collect all vectors in a sparse matrix in rowwise representation without having to store the structures twice. (In the future, the term $g_i(x)$ could be further decomposed into convex, concave and remaining parts, if useful). This form of representation is well suited for most computations and structure analysis and changing algorithms, since it extracts the parts, which most of the time must not be worked on or worked on with special algorithms. In addition trivial relaxations can be computed by simple interval analysis on the terms not fitting the type of relaxation (linear, quadratic, ...). Note, that we will not require the user to enter the problem in such a form. We will use graph walks to construct a DAG structure conforming to the specification above from a computational tree specified in one of the supported modeling languages automatically.

The special DAG structure of the expressions makes it possible to construct evaluation visitors which compute values without doing complete graph walks, and only some visitors may be optimized, while others still use complete walks.

All together the model might be represented as follows:

$$\min f(x) \qquad\qquad s.t. \quad F(x) \in [F]$$

where $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is the collection of all constraints, best decomposed as explained above. The box constraints are special constraints. Probably, it would be best to collect them at the beginning or the end of F.

Every root node of the DAG defines a constraint, and the objective function is the *first* constraint in the vector of root nodes. There are, however, constraints of the form

$$z = f(x, y, ...)$$

which correspond to intermediate nodes because all intermediate variables are sustituted back and incorporated into the DAG structure. Every node can be constrained by bounds $[\underline{b}, \overline{b}]$, which are stored in the interval `f_bounds`. Note that due to that the constraint $xy <= x + y$ will have to be rewritten as

$$xy - x - y \in [-\infty, 0].$$

The two main C++ objects used to construct this DAG, the node and the DAG template, are presented next.

The here presented objects are just a simplified version of the implemented structures which make extensive use of templates in order to achieve independence of allocation schemes and memory models. The real implementation is outlined in Appendix A.

4.1 Expression Node

One node of the expression DAG is represented by the `expression_node` class.

```
class expression_node
{
public:
  unsigned int node_num;
  int expression_type;

  // number of parents and children for fast reference
  unsigned int n_parents, n_children;

  vector<double> coeffs;   // coefficients of the sub_expressions

  additional_info_u meaning; // additional expression info

  interval f_bounds;
  unsigned short is_var;

  semantics* sem;;      // this stores info like convexity,
                        // linearity, separability,...

  expression_node();
  ~expression_node();
};
```

In this class, the `node_number` is automatically assigned by the constructor in such a way that it is unique in the DAG. This number can be used for referencing nodes in a portable way, which is independent of reallocation during copying and graph restructuring.

The vector of `signs` is used to store the signs of the involved child expressions. Thus, e.g., $a - b$ is represented as $a + (-b)$ where the sign of b is stored in the second position of the `signs` vector of the `EXPRINFO_SUM` (see below) node.

The `f_bounds` interval is used to constrain the value of the node.

The unsigned short variable `is_var` defines the number of variables which is represented by this node (for DAG reinterpretation when using intermediate variables). If it is 0, this node does not represent any variable.

The `expression_type` describes the type of operation. There is a predefined list which could be extended if necessary. Additional information needed for the operation is stored in the union `meaning` which is of type `additional_info_u` defined as

```
union additional_info_u
{
  void *p;
  bool b;
  int nn;
  double nd;
  interval ni;
  string* s;
  vector<int>* n;
  vector<double>* d;
  vector<interval>* i;
}
```

One of the vectors is allocated if `meaning_is_alloced` is true.

The following expression types are defined

C++ value	Description	contents of `meaning`
`EXPRINFO_GHOST`	ghost node	in deltas these nodes are a replacement for the nodes where the delta should be inserted. Its node number contains the node number of the node in the DAG and `.b` contains info on edge deletion.
`EXPRINFO_CONSTANT`	constant	constant is contained in `.d` or `.nd`
`EXPRINFO_VARIABLE`	variable vector	index of the variable is contained in `.nn`
`EXPRINFO_SUM`	sum of n expressions	`.nd` contains an additional constant
`EXPRINFO_PROD`	product of n expressions	`.nd` contains an additional constant
`EXPRINFO_MAX`	maximum of n expressions	`.nd` contains minimal value
`EXPRINFO_MIN`	minimum of n expressions	`.nd` contains maximal value
`EXPRINFO_INVERT`	q/x	q is contained in `.nd`
`EXPRINFO_DIV`	division	NA
`EXPRINFO_SQUARE`	$(x+q)^2$	q in `.nd`
`EXPRINFO_INTPOWER`	x^n	n constant integer in `.nn`
`EXPRINFO_SQROOT`	$\sqrt{x+c}$	c in `.nd`
`EXPRINFO_POW`	$(x+p)^y$	p in `.nd`
`EXPRINFO_EXP`	e^{x+p}	p in `.nd`
`EXPRINFO_LOG`	$\log(x+p)$	p in `.nd`
`EXPRINFO_SIN`	$\sin(x+\vartheta)$	ϑ in `.nd`
`EXPRINFO_COS`	$\cos(x+\vartheta)$	ϑ in `.nd`
`EXPRINFO_ATAN2`	$\mathrm{atan}(x,y)$	NA
`EXPRINFO_GAUSS`	$e^{(x-m)^2/s^2}$	m, s in `.d`
`EXPRINFO_LIN`	linear function $A.x$	matrix A in `.p` until the matrix type is fixed
`EXPRINFO_QUAD`	quadratic function $x^\top.G.x$	matrix G in `.p` until the matrix type is fixed.

4.2 DAG base classes

The model is created using general DAG templates, whose default forms are described in this section.

Every DAG consists of a collection of `dag_node`s which have the following form. The vectors `_C_parents` and `_C_children` contain pointers to the parents and the children of the node in the graph. The graph methods make sure that

the vectors are updated correctly.

```
template <class _Tp>
class dag_node
{
public:
  _Tp _C_data;
  // the reason for using void * in the following vectors
  // is connected to the use of templates and the complicated
  // connection between the dag class and the dag_node class
  // maybe it will be possible to change that to dag_node* in
  // the future after revisiting the implementation. Up to now
  // a type cast to dag_node* should be made when using entries
  // of _C_parents or _C_children.
  vector<void *> _C_parents;
  vector<void *> _C_children;

  dag_node();   // a constructor
  ~dag_node();  // a destructor
};
```

Using these nodes, a basic DAG is constructed, which takes care of proper node allocation. The nodes `_C_ground` and `_C_sky` are *virtual* nodes in the graph, which are parent of all roots, and child of all leafs, respectively. They are used for easier graph manipulation only. An empty DAG contains these virtual nodes only and an edge from `_C_sky` to `_C_ground`.

```
template <class _Tp>
class dag_base
{
public:
   dag_node<_Tp>* _C_ground;
   dag_node<_Tp>* _C_sky;
   dag_base();    // a constructor
   ~dag_base();   // a destructor

   dag_node<_Tp>* _C_get_node();           // construct a new node
   void _C_put_node(dag_node<_Tp>* __p);   // destroy a node
}
```

The `dag` class is a subclass of this base class containing methods for DAG manipulation using *walkers*, a generalization of pointers to DAG nodes. Both of these classes are described below.

```
template <class _Tp>
class dag : dag_base<_Tp>   // this is from gtl_dagbase.h
{
```

```
private:
  typedef dag<_Tp> _Self;
  typedef dag_base<_Tp> _Base;
  typedef dag_walker<_Tp> walker;

  // constructors, destructor
  dag();
  dag(const _Self& __dag);
  ~dag();

  // and a lot of methods which usually are not used directly,
  // which are only used by the construction methods in class model.
};

template <class _Tp>
class dag_walker
{
protected:
  // this is the pointer to the current node!
  dag_node<_Tp>* _C_w_cur;

public:
  // constructors and destructor
  dag_walker();
  dag_walker(dag_node* __x);          // points to *__x afterwards
  dag_walker(const dag_walker& __x);  // copy constructor
  ~dag_walker();

  _Tp& operator*();              // returns the data of the node

  // methods for gaining info on the node
  size_type n_children();   // number of children
  size_type n_parents();    // number of parents

  bool is_root();           // points to a root node
  bool is_leaf();           // points to a leaf node
  bool is_ground();         // we are at the ground (i.e. below all roots
  bool is_sky();            // we are in the sky (i.e. above all leafs)

  // navigation (i.e. change to another element. This is a
  // generalization of pointer arithmetic operators ++ and --.
  // << goes up to one of the parents, which can be chosen explicitely,
  // and >> goes down to one of the children)

  // go up to parent __i
  dag_walker operator<<(parents_iterator __i);
```

```
    // go down to child __i
    dag_walker operator>>(children_iterator __i);

    // and the same combined with assignment
    dag_walker& operator<<=(parents_iterator __i);
    dag_walker& operator>>=(children_iterator __i);
  };
```

4.3 Semantics

This class contains data for additional semantic information on the various constraints. The `property_flags` structure contains information on the mathematical properties (linearity, separability, convexity, ...). The `info_flag` contains properties important for the solution algorithms (redundancy, activity, ...). Possibly additional semantics will have to be added during the development of the strategy engine.

Filling the semantic information will be done as far as possible during the initial DAG building process. There will, however, be times when the information changes during the solution process as constraints become redundant or the variables involved in constraints change their domain hereby changing the properties of some constraints.

```
#ifndef _SEMANTICS_H_
#define _SEMANTICS_H_

typedef enum {true, false, maybe} tristate;

class semantics
{
public:
  struct {
    tristate linear;
    tristate quadratic;
    tristate convex;
    tristate concave;
    tristate separable;
    tristate univariate;
  } property_flags;                    // mathematical constraint properti

  struct {
    bool redundant;
    tristate inactive;
    bool deleted;
  } info_flags;                        // algorithmic properties
public:
  // and others needed
```

```
};

#endif // _SEMANTICS_H_
```

5 Evaluators

Most of the inference engines will not need to develop their own evaluation routines for traversing the DAG of expressions. They can make use of evaluator classes, as described below.

As an example a simple function evaluator has been implemented.

The important bits of the evaluator are

```
class func_eval
{
  // __x: where to evaluate
  // __v: what variables have changed since the last evaluation
  // __c: a cache
  func_eval(const vector<double>& __x, variable_indicator& __v,
            vector<vector<double> >& __c);

  vector<double> evaluate();
};
```

and the actual implementation looks somewhat like

```
#ifndef _FUNC_EVALUATOR_H_
#define _FUNC_EVALUATOR_H_

#include <evaluator.h>
#include <expression.h>
#include <eval_main.h>
#include <linalg.h>
#include <math.h>

typedef double (*func_evaluator)(vector<double>* __x,
                                 const variable_indicator& __v);

struct func_eval_type
{
  const vector<double>* x;
  vector<double>* cache;
  double r;
  unsigned int n;
};

class func_eval :
```

```
  cached_forward_evaluator_base<func_eval_type,expression_node,double>
{
private
  typedef cached_forward_evaluator_base<func_eval_type,expression_node,
                                        double> _Base;

protected:
  bool is_cached(const node_data_type& __data)
  {
    if(eval_data.cache && __data.n_parents > 1 && __data.n_children > 0)
      return true;
    else
      return false;
  }

public:
  func_eval(const vector<double>& __x, const variable_indicator& __v,
            vector<double>* __c) : _Base()
  {
    eval_data.x = &__x;
    eval_data.cache = __c;
    v_ind = &__v;
    eval_data.n = 0;
  }

  func_eval(const func_eval& __v) : _Base(__v) {}

  ~func_eval() {}

  void initialize() { return; }

  bool initialize(const expression_info& __data)
  {
    if(__data.ev && __data.ev[FUNC_EVALUATOR])
    // is there a short-cut evaluator defined?
    {
      eval_data.r = (*(func_evaluator)__data.ev[FUNC_EVALUATOR])(eval_data.
                                                                 v_i);
      return false;     // don't perform the remaining graph walk
    }
    else
    {
      switch(__data.expression_type)
      {
        case EXPRINFO_SUM:
        case EXPRINFO_PROD:
```

```
      case EXPRINFO_MAX:
      case EXPRINFO_MIN:
      case EXPRINFO_INVERT:
        eval_data.r = __data.meaning.nd();
        break;
    }
    return true;
  }
}

void calculate(const expression_info& __data)
{
  if(__data.expression_type > 0)
  {
    eval_data.r = __data.f_evaluate(-1, __data.meaning.nn(), eval_data.x,
                                    eval_data.r, 0, NULL);
  }
}

void retrieve_from_cache(const expression_info& __data)
{
  eval_data.r = (*eval_data.cache)[__data.node_num];
}

void update(double __rval)
{
  eval_data.r = __rval;
}

void update(const expression_info& __data, double __rval)
{
  if(__data.expression_type < 0)
  {
    switch(__data.expression_type)
    {
      case EXPRINFO_CONSTANT:
        eval_data.r = __data.meaning.nd();
        break;
      case EXPRINFO_VARIABLE:
        eval_data.r = eval_data.x[__data.meaning.nn()];
        break;
      case EXPRINFO_SUM:
        eval_data.r += __data.coeffs[eval_data.n++]*__rval;
        break;
      case EXPRINFO_PROD:
        // the coefficients MUST be collected in __data.meaning.nd()
```

```
  eval_data.r *= __rval;
  break;
case EXPRINFO_MAX:
  __rval *= __data.coeffs[eval_data.n++];
  if(__rval > eval_data.r)
    eval_data.r = __rval;
  break;
case EXPRINFO_MIN:
  __rval *= __data.coeffs[eval_data.n++];
  if(__rval < eval_data.r)
    eval_data.r = __rval;
  break;
case EXPRINFO_INVERT:
  eval_data.r /= __rval;
  break;
case EXPRINFO_DIV:
  // this evaluator requires, that the second coefficient
  // is put into the first.
  if(eval_data.n == 0)
    eval_data.r = __rval*__data.coeffs[0];
  else
    eval_data.r /= __rval;
  ++eval_data.n;
  break;
case EXPRINFO_SQUARE:
  { double h = __data.coeffs[0]*__rval+__data.meaning.nd();
    eval_data.r = h*h;
  }
  break;
case EXPRINFO_INTPOWER:
  { int h = __data.meaning.nn();
    if(h == 0)
      eval_data.r = 1;
    else
    {
      double k = __data.coeffs[0]*__rval+__data.meaning.nd();
      switch(h)
      {
        case 1:
          eval_data.r = k;
          break;
        case 2:
          eval_data.r = k*k;
          break;
        case -1:
          eval_data.r = 1./k;
```

```
            break;
          case -2:
            eval_data.r = 1./(k*k);
            break;
          default:
            if(h & 1) // is odd
            {
              if(k < 0)
                eval_data.r = -pow(-k, h);
              else
                eval_data.r = pow(k, h));
            }
            else // even
              eval_data.r = pow(abs(k), h);
            break;
        }
      }
    break;
  case EXPRINFO_SQROOT:
    eval_data.r = sqrt(__data.coeffs[0]*__rval+__data.meaning.nd());
    break;
  case EXPRINFO_POW:
    __rval *= __data.coeffs[eval_data.n];
    if(n == 0)
      eval_data.r = __rval+__data.meaning.nd();
    else
      eval_data.r = pow(eval_data.r, __rval);
    ++eval_data.n;
    break;
  case EXPRINFO_EXP:
    eval_data.r = exp(__rval*__data.coeffs[0]+__data.meaning.nd());
    break;
  case EXPRINFO_LOG:
    eval_data.r = log(__rval*__data.coeffs[0]+__data.meaning.nd());
    break;
  case EXPRINFO_SIN:
    eval_data.r = sin(__rval*__data.coeffs[0]+__data.meaning.nd());
    break;
  case EXPRINFO_COS:
    eval_data.r = cos(__rval*__data.coeffs[0]+__data.meaning.nd());
    break;
  case EXPRINFO_ATAN2:
    __rval *= __data.coeffs[eval_data.n];
    if(eval_data.n == 0)
      eval_data.r = __rval;
    else
```

```
            eval_data.r = atan2(eval_data.r, __rval);
          ++eval_data.n;
          break;
        case EXPRINFO_GAUSS:
          { double h = (__data.coeffs[0]*__rval-__data.additional_info.d()[
                                 __data.additional_info.d()[2];
            eval_data.r = exp(h*h);
          }
          break;
        case EXPRINFO_LIN:
          // matrix multiply A x
          break;
        case EXPRINFO_QUAD:
          // matrix multiply x^T A x
          break;
        default:
          // give bad messages
          break;
      }
    }
    else if(__data.expression_type > 0)
      // update the function arguments
      eval_data.r = __data.f_evaluate(n++, __data.meaning.nn(), eval_data.x
                                      v_i, eval_data.r, __rval, NULL);
    // use the cache only if it is worthwhile
    if(eval_data.cache && __data.n_parents > 1 && __data.n_children > 0)
      (*eval_data.cache)[__data.node_num] = eval_data.r;
  }

  const vector<double>& calculate_value(bool eval_all)
  {
    return eval_data.r;
  }
};

#endif /* _FUNC_EVALUATOR_H_ */
```

In order to speed up development of new evaluators, a number of base classes is defined for subclassing.

```
#ifndef _EVALUATOR_H_
#define _EVALUATOR_H_

#include <vector.h>
#include <gtl_dag.h>
#include <g_algo.h>
```

```
class variable_indicator
{
private:
  vector<uint32> v;
public:
  variable_indicator() : v() {}
  variable_indicator(vector<int> __v, int size) : v()

  bool match(const variable_indicator& __v)
};

template <class _Tp, class _NData, class _Result>
class _evaluator_base
{
private:
public:
  typedef data_type _Tp;
  typedef node_data_type _NData;

  typedef return_value _Result;
private:
  _Tp eval_data;

public:
  virtual _evaluator_base();
  virtual _evaluator_base(const _Tp& __x);
  virtual ~_evaluator_base();

  virtual return_value vvalue() {}
  virtual return_value value() {}

  virtual void vcollect(const return_value& __cresult) {}
  virtual void collect(const node_data_type& __data,
                       const return_value& __cresult) {}

  virtual void postorder(const node_data_type& __data) {}

  template <class _Walker>
  result_type evaluate(_Walker __start)
  {
    return recursive_walk(__start, *this);
  }
};

template <class _Tp, class _NData, class _Result>
class evaluator_base : public _evaluator_base<_Tp,_NData,_Result>
```

```
{
public:
  virtual void preorder(const node_data_type& __data) {}
};

template <class _Tp, class _NData, class _Result>
class cached_evaluator_base : public _evaluator_base<_Tp, _NData, _Result>
{
private:
  variable_indicator* v_ind;

public:
  virtual bool preorder(const node_data_type& __data) {}

public:
  virtual cached_evaluator_base();
  virtual cached_evaluator_base(const _Tp& __x, const variable_indicator& _
  virtual ~cached_evaluator_base();
};

template <class _Tp, class _NData, class _Result>
class forward_evaluator_base : public evaluator_base<_Tp, _NData, _Result>
{
public:
  // only virtual constructors and destructors
  virtual forward_evaluator_base();
  virtual forward_evaluator_base(const _Tp& __x);
  virtual ~forward_evaluator_base();

public:
  virtual void initialize();
  virtual void initialize(const node_data_type& __data);
  virtual void calculate(const node_data_type& __data);
  virtual void update(const return_value& __rval);
  virtual void update(const node_data_type& __data, const return_value& __r
  virtual return_value calculate_value(bool eval_all);
};

template <class _Tp, class _NData, class _Result>
class backward_evaluator_base : public evaluator_base<_Tp, _NData, _Result>
{
public:
  // only virtual constructors and destructors
  virtual backward_evaluator_base();
  virtual backward_evaluator_base(const _Tp& __x);
  virtual ~backward_evaluator_base();
```

```
public:
  virtual void initialize();
  virtual void initialize(const node_data_type& __data);
  virtual void calculate(const node_data_type& __data);
  virtual void update(const node_data_type& __data, const return_value& __r
  virtual void update(const return_value& __rval);
  virtual return_value calculate_value(bool eval_all);
};

template <class _Tp, class _NData, class _Result>
class cached_forward_evaluator_base : public cached_evaluator_base<_Tp, _ND
{
public:
  // only virtual constructors and destructors
  virtual forward_evaluator_base();
  virtual forward_evaluator_base(const _Tp& __x);
  virtual ~forward_evaluator_base();

public:
  virtual void initialize();
  virtual bool initialize(const node_data_type& __data);
  virtual void calculate(const node_data_type& __data);
  virtual void retrieve_from_cache(const node_data_type& __data);
  virtual void update(const node_data_type& __data, const return_value& __r
  virtual void update(const return_value& __rval);
  virtual return_value calculate_value(bool eval_all);
};

template <class _Tp, class _NData, class _Result>
class cached_backward_evaluator_base : public cached_evaluator_base<_Tp, _N
{
public:
  // only virtual constructors and destructors
  virtual cached_backward_evaluator_base();
  virtual cached_backward_evaluator_base(const _Tp& __x);
  virtual ~cached_backward_evaluator_base();

public:
  virtual void initialize();
  virtual bool calculate(const node_data_type& __data);
  virtual void cleanup(const node_data_type& __data);
  virtual void retrieve_from_cache(const node_data_type& __data);
  virtual void update(const node_data_type& __data, const return_value& __r
  virtual void update(const return_value& __rval);
  virtual return_value calculate_value(bool eval_all);
```

```
};

#endif /* _EVALUATOR_H_ */
```

A The graph template library (GTL)

In this and the following appendices, the basic structure of the graph template library (GTL) used to construct the DAGs is outlined.

A.1 Standard template library (STL)

The following C++ classes from the standard template library (STL) will be used throughout this document in order to simplify develompent of new objects. For a thorough description of the STL see [Stepanov and Lee, 1995], [RPI, 1994] or [Weidl, 1996].

The basic idea of the STL is to reduce implementation time and effort by decomposing algorithmic problems into their basic components. The STL provides parametrized *Containers* for keeping data of arbitrary types, and generic *Algorithms* working on containers. The interface between the Containers and the Algorithm works by a concept called *Iterators*, an abstraction of the algorithm-access to the containers and the objects stored in them.

Unfortunately, the STL only provides *sequence containers* (linear ones like lists, queues, and vectors) or *associative containers* like sets and maps. More complicated structures like trees are not provided, except for binary trees.

Since we will need a lot of tree structures in the optimization algorithm, I thought it would be best to design a template tree structure and an associated access concept, I called *Walkers*.

Sequence Container Classes

Sequence containers are specifically designed to hold sequences of other objects. STL provides three sequence containers: lists (list), arrays (vector), and double ended queues (dqueue). They differ mainly in the amount of storage needed and the overhead for performing certain actions. The following table gives an overview on the effort:

Container	insert/erase overhead at the beginning	in the middle	at the end
vector	linear	linear	amortized constant
list	constant	constant	constant
dequeue	amortized constant	linear	amortized constant

Certain operations can be performed on containers in addition to inserting and deleting entries at every position, e.g. getting the number of stored objects (size).

A large amount of algorithms are designed to work with sequence containers (sort, linear and binary search, and there is the strength of the STL concept. These algorithms are implemented in a generic way, and the interface algorithm/container works by the *iterator* concept (see A.3). In the case of vectors, iterators are just integers indexing into the array, in the case of lists an iterator is a more elaborate object. By using iterators, every element of the container can be accessed in a well-defined way.

Associative Container Classes

Associative containers are designed to hold key–value pairs, where keys and values are objects. The only restriction is that there has to be an ordering relation on the set of keys (operator¡). STL provides four sequence containers: sets (set), multivalued sets (multiset), maps (map), and multikeyed maps (multimap). The difference between the multi- and the ordinary containers is as follows. In a set or map a given key may appear only once, whereas in a multiset or multimap a given key need not be unique. The difference between sets and maps is that sets contain their key (i.e. the value object and the key object are the same), while for maps key and value objects differ.

As for sequence containers, the operations are inserting, deleting, ... and iterators provide the interfaces to the algorithms.

A.2 Graph Classes

The new class templates defined in `ntree.h`, `dag.h`, and `graph.h`and the included internal header files `gtl_` ... provide various graph templates for arbitrary node types (graphs, labelled graphs, directed graphs, directed acyclic graphs, and forests (n-ary multirooted trees) — WARNING: Not all of them are fully implemented). The templates start roughly like

```
template <class _Tp,
  template <class __Ty, class __AllocT> class _Ctr = vector,
  class _PtrAlloc = __STL_DEFAULT_ALLOCATOR(void *),
  class _Alloc = __STL_DEFAULT_ALLOCATOR(_Tp) >
class ctree
...
```

that means, that graphs of arbitrary types (`_Tp`) can be constructed. The children (eventually the parents) of every node will be managed through a container class (`_Ctr` - by default vector - i.e. an array)[1]. The `_PtrAlloc` and `_Alloc` classes are used for allocation method independence, and the default can usually be kept as is.

As an example, we could define a forest of integers simply by

[1] Note that in the first implementation only sequence container classes can be used. For associative containers there will have to be an additional implementation, since the container class would depend on a data type and a key type for maps and a comparison operator

```
ntree<int> itree;
```

and a directed acyclic graph of `expression` nodes by

```
dag<expression> expression_graph;
```

If many insertions and deletions of subtrees in the middle are to be made, then a vector would be a bad choice for the container class holding the nodes of the subtrees, since inserting into the middle of a vector needs reallocation and copying. We should instead take a list to manage the children, and the definition would change to

```
ntree<int,list> itree;
```

Except for various constructors and the destructor a few methods are provided for handling graphs. Here, we will often mention walkers. They are described in more detail in A.3.

`empty` returns a boolean, true if the graph does not contain any regular node.

`max_size` the maximum number of nodes the graph container can store. This is unlimited except for the amount of memory available.

`clear` erases the whole graph.

For **Trees** additional methods are available:

`insert` at the walker position, a node or a subgraph are inserted.

`push_child` adds a new child to the node pointed to by the walker (it is added at the back of the container).

`push_subtree` adds a tree as subtree at the node pointed to by the walker.

`insert_child, insert_subtree` these work like `push_child` and `push_subtree`, except that they are also provided an iterator of the children-container. The child, respectively subtree, is inserted there and not as last child.

`erase` deletes one node from the tree, connecting all children to the parent node.

`erase_subtree` erases the subtree whose root is the walker position provided.

`pop_child, pop_subtree, erase_child` are the inverse operations to `push_child`, `push_subtree`, and `insert_child`.

`depth` returns the level in the tree of the node the walker points to.

Each **tree node** provides in addition an element of `ctree_data_hook`, where data could be stored during iterative tree walks. Evaluation routines could e.g. make use of these data hooks. Note that this concept does not apply to the other graph types.

For **directed graphs** the additional constructor methods are:

`push` adds a new node between parents and children, hereby erasing all existing links between parents and children. In the containers managing the links, the new edge is added at the end.

`push_between` adds a new node between a parent and a child leaving existing links between them intact.

`push_front` adds a new node between parents and children, hereby erasing all existing links between parents and children. In the containers managing the links, the new edgees are added at the beginning.

`push_front_between` adds a new node between parents and children leaving existing links between them intact.

`insert` adds a new node between parents and children, hereby erasing all existing links between parents and children. In the containers managing the links, the new edge is added at the specified iterator position.

`insert_between` adds a new node between a parent and a child leaving existing links between them intact.

`add_edge` This adds an edge between two nodes.

`remove_edge` This removes an edge between two nodes.

`insert_subgraph` With this method, a complete subgraph is added between lists of parents and children.

A.3 Iterators and Walkers

Iterators

Iterators, the interface between container classes and algorithms, are a generalization of pointers. They have a dereference operator * defined which returns the iterator's data type.

There are five iterator categories, connected as depicted in figure 4. An arrow in this diagram points from the special iterators to the more general ones. All STL container classes provide at least bidirectional iterators. Output iterators can be used to generate output (they can be written to only), input iterators for input (they can be read from only). Both can be incremented and are suited for *single-pass, one-directional algorithms.*

Forward iterators can also be incremented only (one way street). They, however, are input and output iterators, so they can be both written to and read from. They may be used to implement *multi-pass one-directional algorithms.*

The next specialization are bidirectional iterators, which can be both incremented and decremented (i.e. they can pass through the container in both directions), and therefore provide just enough flexibility for *general multi-pass algorithms* like bubble sort.

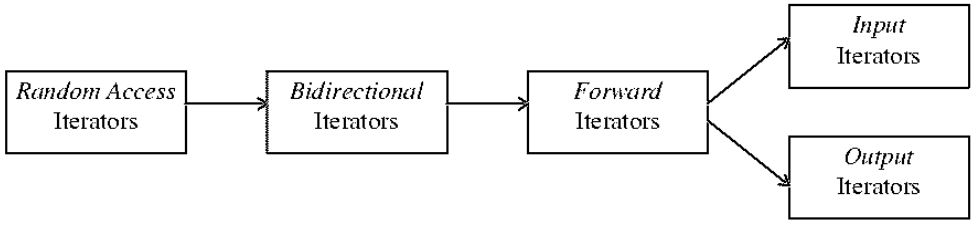

Figure 4: Relation of the iterator categories

The most specialized iterator class are random access iterators. They work like array indices and can be chosen arbitrarily between the start and the end indices of the container. Quicksort or binary search are algorithms which make use of random access iterators.

Iterators can be generated by container methods, and the most important ones are `begin()` (the iterator pointing to the first element of the container) and `end()` (the iterator pointing *beyond* the last element of the container).

Walkers

Iterators cannot capture the full structural advantage of graphs, so for these an adapted concept should be introduced, a *walker*. Since graphs can be walked in many ways, walkers come in different flavors. For general graphs, only recursive walkers can be defined; however for trees we distinguish two basic types, and three different flavours for the second type (the iterative walkers).

The simplest walker category are the *recursive walkers*. Only two operators are defined for them: `<<` goes one level up to a parent node, and `>>` goes one level down to a child. These operators take a pointer to the child or parent (key or iterator) as second argument. Recursive graph walks (see section A.4) are best fitted for them. For non-directed general graphs the walker only provides a `>>` operation.

The other walker category, the *iterative walkers* are best suited for non-recursive **tree walks**. They come in various flavors and the following flags influence their behavior. In addition to the operators defined for recursive walkers, `++`, `--` define transition to the next and previous nodes in the tree walk, respectively. The operator `~` switches from pre-order to post-order visit and vice versa.

node order (aka `order`) This flag can take three values, defined in an enum type `walker_type`. `pre` defines a *preorder* walker (the node is visited before the children are visited), `post` defines a *postorder* walker (children are visited before the parent node), and `pre_post` defines a combined

walker, which visits every node twice. First comes the preorder visit (`w.in_preorder()` method returns `true`), then the children are visited, and afterwards the parent node is visited again (the postorder visit, where the `w.in_preorder()` method returns `false`).

walk direction (aka `left_to_right`) This flag takes boolean values, and if it is `true`, the children are visited "from left to right" (i.e. from `.begin()` to `.end()` for the container class holding the children). If the flag is `false`, the children are visited from "right to left".

depth handling (aka `depth_first`) This flag also takes boolean values. If set to `true`, a *depth first* walk is performed. This means that the walker visits the children of a node before the next node in the same depth level is visited. If the flag is set to `false`, a *breadth first* walk is performed, i.e. all nodes in one level are visited before the walker advances to the next level. (Breadth first tree walks are e.g. used in strategy games; usually depth first walks are preferred because of much smaller memory overhead.)

In addition, walkers can be compared with each other using `==` and `!=`. Like with iterators, equality requires that the walkers are constructed for the same graph, point to the same node, are of the same kind (recursive or iterative including all possible flavors) and are in the same state (preorder or postorder pass).

All walkers provide hooks to the graph's data. The methods below can be applied to every walker of every category.

node Return value for `node` is a pointer to the tree node the walker points to.

n_children returns the number of children a node possesses (for directed graphs and trees).

n_parents returns the number of parents a node possesses (for directed graphs and trees).

n_neighbors returns the number of neighbors a node possesses (for undirected graphs).

data_hook This method returns a reference to a **tree node's** data hook, only interesting for iterative tree walks.

parent_data_hook Similar to the above, this method returns a reference to the parent node's data hook.

child_begin, child_end These methods return iterators to the beginning and to the end, respectively, of the container containing a node's children.

parent_begin, parent_end These methods return iterators to the beginning and to the end, respectively, of the container containing a node's parents.

for_each_child takes a function object as its argument, which is iteratively applied to all the node's children.

for_each_parent takes a function object as its argument, which is iteratively applied to all the node's parents.

Walkers are constructed from tree classes using various methods. The `begin` method takes three arguments (the flags from above), where the default values are `order = pre_post`, `left_to_right = true`, `depth_first = true`. It returns a walker pointing to the first node visited during the appropriate tree walk.

For all graphs which are implemented using virtual roots (the ground) or virtual leafs (the sky) the methods `ground` and `sky` return walkers to the respective virtual nodes (this includes all sorts of trees, dgraph (directed graph), dag, and graph).

Corresponding to `begin`, the `end` method takes the same arguments and returns a walker pointing beyond the end of the **tree** walk.

Thus, `begin` and `end` work like for iterators, but there are two additional methods. The `ground` method returns the position before the first visited node (the imaginary top node of the tree). This is mainly used for the `insert`, `push_child`, and `push_subtree` methods for constructing the tree. Every tree has a `ground`, even if it does not contain a node. Semantics: `begin() = ++ground()`.

In addition to `end` there exists a `through` method. It takes no arguments and returns the position after the postorder visit of the imaginary root node. Semantics: `through() = ++end()`. The through position is used as a safeguard against walks beyond the tree. We have `++through = through`.

A.4 Algorithms

As for sequence and associative containers, a certain number of algorithms is designed for working with trees. The abstraction from the specific tree class works by making the walkers be the only connection between the tree container and the algorithms.

At first, all algorithms for containers can in principle be used for trees, too. These algorithms can either make use of the iterators associated to every tree class (essentially a depth-first, post-order, left-to-right iterative walker) or can due to their template-like nature be directly provided with a walker. The STL–tutorial [Weidl, 1996] and the STL–documentation [Stepanov and Lee, 1995] provide information on that.

In addition, a number of additional tree-only algorithms is developed, providing a means to perform your favorite tree walk in a general algorithmic fashion. Find below a list of all generic tree algorithms. Here, an `IterativeWalker` is an iterative walker, a `PrePostWalker` is a combined preorder–postorder iterative walker, and `Walker` is any walker (recursive or iterative).

A *visitor object* is a class providing the following methods:

`return_value` The type of the return value of the visitor.

`vinit()` is called instead of `preorder` at a virtual node

`vvalue()` This returns the return value of the visitor at a virtual node.

`value()` This returns the return value of the visitor.

`collect(value_type data, return_value value)` This method is called after a subgraph walk for a child is completed. Provided is the data of the active node and the return value of the subtree walk.

`collect(return_value value)` This method is called at a virtual node after a subgraph walk for a child is completed. Provided is the data of the active node and the return value of the subtree walk.

`preorder(value_type data)` Before the nodes of the subgraph are visited (i.e. in the preorder phase of the graph walk) this method is called. The data provided is the data of the active node.

`postorder(value_type data)` This method is like `preorder` but the method is called after all the return values of the subgraph walks have been collected by `collect`.

`analyze(walker w)` This method is called in general graph walks to decide whether the walk should be continued.

`walk_up(value_type data)` This method decides whether the next node is a parent or a child in a general directed graph walk.

`up()`, `down()` This method returns the next node in directed graph walks to be visited.

`next()` This method returns the next node for undirected graph walks.

A visitor does not need to provide both `preorder` and `postorder` unless it is used in a recursive walking algorithm, which needs pre- and postorder node visits.

The following generic algorithms for iterative tree walkers are provided:

`walk(IterativeWalker first, IterativeWalker last, Function f)` This performs a tree walk using an iterative walker, calling `f` at every visited node. `f` takes four arguments: (a reference to) the node's data, a reference to the node's data hook, a child data iterator range (begin, end). A child data iterator iterates through the data hooks of all children of this node. The function returns `f`.

`pre_post_walk(PrePostWalker first, PrePostWalker last, Function f)` This generic algorithm performs a tree walk, where every node is visited twice (in preorder and in postorder). `f` is like in `walk`, except that it takes a `bool` valued fifth argument. This argument will be `true` in the preorder call and `false` in the postorder call. The return value is `f`.

`pre_post_walk(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2)` A variant of the above, this algorithm performs the same walk, but calling two different functions `f1` and `f2` taking the same arguments as `f` in `walk`. `f1` is called in the preorder visit, and `f2` in the postorder visit. The return value is `f2`.

`var_walk(PrePostWalker first, PrePostWalker last, Function f)` This function performs a tree walk like in `pre_post_walk`. The function `f` has to return a value, which is convertible to `bool`. If the return value of `f` is `true`, the walker is immediately switched from preorder to postorder or vice versa.

The effect of that is the following. If the walker is in the preorder phase visiting the node and `f` returns `true`, then the subtrees of the node are never visited, and visiting of this node is completed in one step. This is useful e.g. for evaluation algorithms, which have a cache or another faster way of computing the values of the subtrees.

If the walker were in the postorder phase, when `f` returned `true`, then the subtrees of this node are visited again, and the node will be reached in the postorder phase again.

`var_walk(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2)` This is like the first variant of `cached_walk`, except that two different functions will be called.

`walk_if(PrePostWalker first, PrePostWalker last, Function f, Predicate pred)` This tree walk works like `var_walk`, except that it takes a further argument, a function object working as predicate. This predicate should return `true` or `false` given the node's data. The return value of the predicate then decides, whether the walkers state will be switched.

`walk_if(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2, Predicate pred)` This `walk_if` variant calls two different functions in the preorder and postorder visits, respectively.

`walk_if(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2, Predicate1 pred1, Predicate2 pred2)` This variant of `walk_if` calls two different functions and two different predicates.

`cached_walk_if(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2, Predicate pred)` A cached walk works like `walk_if`, except that the predicate is evaluated only in the preorder phase, allowing the tree walk to be shortened.

`multi_walk_if(PrePostWalker first, PrePostWalker last, Function1 f1, Function2 f2, Predicate pred)` This generic algorithm is a `walk_if`, which only evaluates the predicate in the postorder phase, making it possible to construct multi pass tree walks.

`walk_up(Walker w, Function f)` If a tree walk upwards to the root node is necessary, use `walk_up`. This function visits the starting node and its parents until the root node of the tree is reached (the true root of this branch of the multirooted tree, not the virtual root node). For every node, the function is called, and `f` is returned in the end.

`var_walk_up(Walker w, Function f)` This algorithm walks up until the root node is reached or the function `f` returns `true`.

`walk_up_if(Walker w, Function f, Predicate pred)` This algorithm works like `var_walk_up`, except that the predicate decides, whether the walk is stopped and not the return value of `f`.

The following generic recursive walk procedures work for all acyclic graphs (forests, dags)

`recursive_preorder_walk(Walker w, Visitor f)` A recursive preorder walk is a tree walk, in which the visitor `f` visits every node during the preorder phase. The algorithm is constructed in a recursive way using a visitor class, not touching the data hook of the tree.

`recursive_postorder_walk(Walker w, Visitor f)` This generic algorithm works like the last one, except that preorder has to be replaced by postorder.

`recursive_walk(Walker w, Visitor f)` In this similar generic algorithm, both preorder and postorder visits are performed.

`recursive_walk_up(Walker w, Visitor f)` Starting from the node the walker points to, the tree is walked upwards until the root node is reached. This is done recursively, so the stack will be unwound after the root node is reached. The visitor method `preorder` is called prior to walking one level up. Then `collect` is used to collect the return value of the recursive call, and afterwards `postorder` is invoked.

`recursive_preorder_walk_up(Walker w, Visitor f)` This is a variant of `recursive_wa` except that the call to `postorder` is omitted.

`recursive_postorder_walk_up(Walker w, Visitor f)` This is a variant of `recursive_walk_up` except that the call to `preorder` is omitted.

... `_if` To any of the recursive algorithms, either by providing a `bool` return value for the visitor methods `preorder` and `postorder` or by adding a predicate to the algorithm call, an `_if` can be appended. The algorithm then works similar to `walk_if`. If the predicate (or the visitor method) return `true`, either the subtrees are omitted or the node is revisited after the subtree walk has been completed.

`recursive_cached_walk(Walker w, Visitor f)` This is a variant of `recursive_walk_if` which only evaluates the predicate in the preorder phase. This generates an algorithm working like `cached_walk_if`.

`recursive_multi_walk(Walker w, Visitor f)` This is a variant of `recursive_walk_if`, which only evaluates the predicate in the postorder phase. This generates an algorithm working like `multi_walk_if`.

Finally, for all graphs there are the generic walk algorithms `general_walk`, `recursive_general_walk` for any graph, and `general_directed_walk`, `general_directed_` `general_directed_walk_up`, `recursive_general_directed_walk`, `recursive_general_di` `recursive_general_directed_walk_up` for all directed graphs.

Bibliography

[RPI, 1994] The standard template library online reference. http://www.cs.rpi.edu/projects/STL/stl/stl.html, Rensselaer Polytechnic Institute, 1994.

[Stepanov and Lee, 1995] A. Stepanov and M. Lee. The standard template library. ftp://butler.hpl.hp.com/stl/stl.zip, Hewlett–Packard, 1995.

[Weidl, 1996] J. Weidl. The standard template library tutorial. http://www.infosys.tuwien.ac.at/Research/Component/tutorial/prwmain.htm, 1996.