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 4

Changes and new features in API 2.x

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

1. Introduction

The API 1 (its newest version being 1.32) has shown a few weaknesses during the test and implementation phase. So, it was decided to produce a full version upgrade.

The most important changes are *name changes* (see Section 2), *changes in the file structure* (see Section 3), *additional features* (see Section 4.3), and *changes due to redisign* (see Section 5).

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

2. Name changes

- In inference engines `__model` was renamed to `__wnode`.
- In expressions `expression_type` was renamed to `operator_type` and `meaning` was renamed to `params`.

3. Changes in the file structure

- There is now a top-level `Makefile` recursively calling the Makefiles of the `api`, `report`, `management`, and all `module-` directories.
- The `control_data` part of the `inference_engine` files was put into the files `control_data.h`, `control_data.cc`, and `cdat-inline.h`. This was necessary because report and management modules need the `control_data` class, as well.
- The `search_node`, `full_node`, `delta_node`, and `work_node` classes were taken out of `search_graph.h` and put into `search_node.h` and `search_node.cc`, reducing compile time because inference engines no longer need to include `search_graph.h` but it suffices to include `search_node.h`. Note that this change might cause some `namespace std` problems, but these can easily be corrected by specifying.

  ```
  using namespace std;
  ```

  somewhere near the top of the inference engine files.

  The files `search_graph.hpp` and the remaining part of `search_graph.h` were merged and then split into `search_graph.h` and `search_graph.cc`, containing most of the (previously inlined) big functions. `sum_deltas.hpp` was renamed to `sum_deltas.h`.

- The `enum tristate` was taken out of `semantics.h` and put into the new header file `coconut_types.h`.

4. Additional features

Some new things have been implemented.

4.1. Expressions. The list of expressions was strongly enhanced, and it contains the following `operator_types`. Every child of an expression is multiplied by its corresponding entry in the `coeffs` vector before it is used in the expression. So the value x_i used in the expression definitions below already contains the factor from the `coeffs` array. If the value y_i is used in the description, you can safely assume that the `coeffs`-entry is 1.

There is a number of expressions with real result:

`EXPRINFO_GHOST`: This is a virtual expression needed for `dag_delta`s.
`EXPRINFO_CONSTANT`: The value of the constant is contained in `params.nd()`.
`EXPRINFO_VARIABLE`: The variable number is contained in `params.nn()`.
`EXPRINFO_SUM`: This is a general linear combination of the form

$$\sum_{i=1}^{n} x_i + c$$

where c is stored in `params.nd()`.
`EXPRINFO_MEAN`: This is a sum node where it is guaranteed that all coefficients are positive, and their sum is 1. So this is a convex linear combination without constant term:

$$\sum_{i=0}^{n} x_i.$$

`EXPRINFO_PROD`: The product node has the form

$$c \prod_{i=0}^{n} y_i$$

where c is in `params.nd()`.
`EXPRINFO_MAX`: The maximum node is defined as

$$\max(c, x_i \quad i = 0, \dots, n)$$

with c in `params.nd()`.
`EXPRINFO_MIN`: The maximum node is defined as

$$\min(c, x_i \quad i = 0, \dots, n)$$

with c in `params.nd()`.
`EXPRINFO_MONOME`: The general monomial node is defined as

$$\prod_{i=0}^{n-1} x_i^{n_i}$$

where the n_i are contained in the vector `params.n()`.

`EXPRINFO_SCPROD`: The scalar product is defined as

$$\sum_{i=0}^{k-1} x_{2k}x_{2k+1}$$

without parameters.

`EXPRINFO_NORM`: The norm node has the form

$$\|x\|_k$$

where k is stored in `params.nd()`, and x is the vector built from the values of all children.

`EXPRINFO_INVERT`: Inversion is defined as

$$a/y_0$$

with a in `params.nd()`.

`EXPRINFO_SQUARE`: The square node is

$$(x_0 + q)^2$$

where q is stored in `params.nd()`.

`EXPRINFO_SQROOT`: The square node is

$$\sqrt{x_0 + q}$$

where q is stored in `params.nd()`.

`EXPRINFO_ABS`: The absolute value node is

$$|x_0 + q|$$

where q is stored in `params.nd()`.

`EXPRINFO_INTPOWER`: The integer power node is

$$x_0^n$$

where the integer n is in `params.nn()`.

`EXPRINFO_EXP`: The exponential node is

$$e^{x_0+q}$$

where q is stored in `params.nd()`.

`EXPRINFO_LOG`: The logarithm node is

$$\log(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_SIN`: The sine node is

$$\sin(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_COS`: The cosine node is

$$\cos(x_0 + q)$$

where q is stored in `params.nd()`.

`EXPRINFO_GAUSS`: The gaussian function node is

$$e^{-(x_0+q_0)^2/q_1^2}$$

where the two dimensional vector q is stored in `params.d()`.

`EXPRINFO_POLY`: The polynomial node is

$$\sum_{i=0}^{n} \alpha_i x_0^i$$

where the coefficients α_i are stored in `params.d()`.

`EXPRINFO_POW`: The general power node is

$$(x_0 + p)^{x_1}$$

with p in `params.nd()`.

`EXPRINFO_DIV`: The general division node is

$$a * y_0/y_1$$

and a is stored in `params.nd()`.

`EXPRINFO_ATAN2`: The atan2 node has no parameters, and its definition is

$$\mathtt{atan2}(x_0, x_1) = \operatorname{atan}(x_0/x_1).$$

`EXPRINFO_LIN`: This node is a linear combination of **all variables**. In `params.nn()` the row number in the matrix of linear constraints is stored.

`EXPRINFO_QUAD`: This node is a quadratic form in **all variables**. The `params.m()` stores the enhanced matrix $(A|b)$ for the function

$$x^T A x + b^T x$$

A is assumed symmetric and sparse.

`EXPRINFO_IF`: We get the interval $\mathbf{x}$ from `params.ni()` and define

$$n = \begin{cases} x_1 & \text{if } x_0 \in \mathbf{x} \\ x_2 & \text{otherwise.} \end{cases}$$

The next operators have discrete results:

`EXPRINFO_IN`: The point defined by $x = (x_i)$ all children is tested against the box $\mathbf{x}$ stored in `params.i()`. If $x \in \int(\mathbf{x})$ the value is 1, if $x \in \partial\mathbf{x}$, the return value is 0, and if $x \notin \mathbf{x}$ the node's value is -1.

`EXPRINFO_AND`: The value is 1 if all $x_i \in \mathbf{x}_i$ where the $\mathbf{x}_i$ are stored in `params.i()`, and 0 otherwise.

`EXPRINFO_OR`: The value is 0 if all $x_i \notin \mathbf{x}_i$ where the $\mathbf{x}_i$ are stored in `params.i()`, and 1 otherwise.

`EXPRINFO_NOT`: The value is 0 if $x_0 \in \mathbf{x}$ and 1 otherwise. Here $\mathbf{x}$ is in `params.ni()`.

`EXPRINFO_IMPLIES`: The value is 1 if the following is true:

$$x_0 \in \mathbf{x}_0 \implies x_1 \in \mathbf{x}_1$$

where the $\mathbf{x}_i$ are stored in `params.i()`. Otherwise, the value is 0.

`EXPRINFO_COUNT`: The count node returns for how many i we have $x_i \in \mathbf{x}_i$, where the intervals $\mathbf{x}_i$ are stored in `params.i()`.

`EXPRINFO_ALLDIFF`: This node returns 1 if all children are pairwise different. Here different means

$$|x_i - x_j| > \delta$$

with δ in `params.nd()`.

`EXPRINFO_HISTOGRAM`: The histogram node counts how many children x_i are in which interval $\mathbf{x}_{2k}$. The numbers n_k are collected, and if for all k the value $n_k \in \mathbf{x}_{2k+1}$, the result is 1. Otherwise, it is 0.

`EXPRINFO_LEVEL`: The matrix A in `params.mi()` contains a set of boxes $A_{i:}$ contained within another. The result is the smallest row index j such that the vector $x = (x_i)$ is contained in $A_{j:}$.

`EXPRINFO_NEIGHBOR`: This returns 1 if x_0 and x_1 are neighbors, i.e., if (x_0, x_1) is in the list of allowed value pairs (v_{2k}, v_{2k+1}) where the vector v is stored in `params.n()`. The x_i and v_i are integers.

`EXPRINFO_NOGOOD`: This returns 1 if $x \neq v$ with v in `params.n()`. Both $x = (x_i)$ and v are integer vectors.

There are two integration nodes:

`EXPRINFO_EXPECTATION`: integrating over all variables in the subgraph over all $\mathbb{R}$.

`EXPRINFO_INTEGRAL`: integrating over all variables in the subgraph, for which the integration area defined in `params.i()` is a non-empty interval. Variables with empty interval are considered free.

The following highly complex operations implicitly build matrices all use params.nm to hold the parameters. Special is params.nm[0], because it contains the noted info. If the matrix is sparse, the remaining rows contain column index information.

`EXPRINFO_DET`: The determinant of the matrix formed by the child nodes. `params.nm()(0,0)` specifies whether the matrix is dense (1) or sparse (0), `params.nm()(0,1)` specifies the dimension of the matrix.

`EXPRINFO_COND`: The condition number of a matrix formed by the child nodes. `params.nm()(0,0)` specifies whether the matrix is dense (1) or sparse (0), `params.nm()(0,1)` and `params.nm()(0,2)` specifies the dimension of the matrix, and `params.nm()(0,3)` deterimines which condition number is calculated. Here 0 is for ∞.

`EXPRINFO_PSD`: This is 1 if the matrix is positive semidefinite and 0 otherwise. The `params` are as in `EXPRINFO_DET`

`EXPRINFO_MPROD`: The value of this node is

$$||Ax - b||_2^2$$

where the nodes define sequentially A, b, x, and the `params` define the matrix dimensions as in `EXPRINFO_COND`.

`EXPRINFO_FEM`: The value of this node is

$$||A^T DAx - b||_2^2$$

where the nodes sequentially define A, D, b, x, and the `params` define the matrix dimensions as in `EXPRINFO_COND`.

Finally, there are two nodes containing constant matrices:

`EXPRINFO_CMPROD`: The node returns the value

$$||Ax - b||_2^2$$

where the matrix $(A|b)$ is stored in `params.m()`.

`EXPRINFO_CGFEM`: The node returns the value

$$||A^T DAx - b||_2^2$$

where the matrix (A/b) is stored in `params.m()`, and the nodes sequentially define D, x.

4.2. Semantics. There are new information bits in the semantics structure which are needed for the new expressions and the MIP relaxation.

`integer`: The methods

```
bool integer_flag() const;
void integer_flag(bool __c)
```

are used to set/reset and query it. If set, that means that the node can take only integer values. For variable nodes this effectively means that the variable is an integer variable.

`hard`: The methods

```
bool hard_flag() const;
void hard_flag(bool __c)
```

are used to set/reset and query it. If set, that means that the node (and the flag is only defined for constraints) is a hard constraint. If unset, the constraint is a soft constraint.

4.3. Small additions.

- The `additional_info_u` type now also contains `unsigned int` and `vector<unsigned int>` entries, accessible by the methods `nu()` and `u()`, respectively.
- Evaluators now do no longer perform cached walks but cached short-cut walks through the DAG. This needed the change of the return values of some internal evaluator methods. For evaluators this meant changing the `initialize` and `update` methods.
- The `ie_return_type` was changed in two ways. First it was adapted to suit the redesign of the `delta` system (see Section 5.2). The most important consequence of this was the change of the `get` method to

```
ie_return_type::get(work_node& wn, double thresh);
```

Second, actually a change already performed in API 1.31, there is an additional `information` part in the `ie_return_type`. This can be set by the inference engine to report back information to the strategy engine without need to pack this information into meaningless deltas. E.g., the `hard variables` service of the `basic_split` inference engine uses this method to report the list of presumed hard variables. For the management of this information new methods

```
bool set_information(const std::string& iname, const additional_info_u& a,
                     bool force = false);

template <class _C>
bool set_information(const std::string& iname, const _C& a,
                     bool force = false);

const additional_info_u& information(const std::string& iname) const;

const additional_info_u& information(const char* iname) const;
```

have been added for setting and retrieval of the information. Information bits are indexed by `strings`.

- There is a new constructor for the `model` class taking an `erased_part` and producing a new `model`. This is needed in `dag_delta` if parts of the DAG are removed. The `erased_part` is stored to make unapplying the DAG delta possible.
- The `model` simplifier was improved, as it now can also simplify `dag_deltas` containin `ghost` nodes.
- There are now methods to collect the coefficients of a linear expression.

```
bool get_linear_coeffs(const walker& expr, sparse_vector<double>& coeffs,
                   double& constant, const std::vector<interval>& _ranges);
bool get_linear_coeffs(const walker& expr, sparse_vector<double>& coeffs,
                       double& constant);
```

They return whether the expression was indeed linear. The `coeffs` and `constant` contain coefficients and constant term. The first method checks whether the expression is linear of restricted to the box given by `_ranges`.

- The `read` and `write` methods of `model` and `expression` now can read and write matrices.
- The `interval.h` header file now contains macros `is_integer` and `get_integer` for checking whether a `double` actually contains an integer, and for retrieving that integer value as `int`.
- In `interval.h` there is an additional function `gainfactor` comparing two intervals and computing a safeguarded quotient of their widths.
- The file `linalg.h` now contains functions for multiplying `MTL` matrices with standard `std:vector` vectors.
- All VGTL now is in namespace `vgtl` and no longer in namespace `std`.

5. Changes due to redesign

Some parts of the system had a bad design in the API 1. So they have been redesigned for various reasons.

5.1. Anotations. The class `annotation` is gone in its previous form. Now the annotations are kept as entries into the search database. Every single annotation represents one row from one table in the search database.

The list of annotations valid in a work node are used to generate a database `view` for all inference engines working on the work node, restricting the database to the entries valid for this work node only.

5.2. Deltas. In the previous API it was unsafe to keep more than one `work_node` because deltas could not be used twice. Since a `delta` stored all information needed for undoing it, it could not be applied twice safely, not even to different work nodes.

Now a `delta` only contains the information needed for **applying** it, and it does no longer contain a method for **unapplying**. Instead, the `apply` method constructs an `undelta` object containing the `unapply` method and its information. This `undelta` is stored in the `work_node` in a map linking it to the `delta_id` of the `delta` it was constructed from.

The `delta`s now are stored in the search database in the `"deltas"` table. The `row_id` in this table is used as `delta_id`. So the `work_node` no longer has to keep track of `delta_id`s.

Every `delta` is kept by a `search_node` in the `search_graph`, and is removed from the search database if its keeping node is destroyed. If a delta has to be kept although a node needs to be destroyed, it must be promoted to the parent node before.

If the `get` method of `ie_return_type` is called and a `delta_id` is returned, this `delta` is before returning its `delta_id` stored in the search database. Before it is stored, the `convert` method of the `delta` is called. E.g., the `point_delta` adds a new entry in the `"point"` table of the search database and converts itself to an `annotation_delta`, which is itself stored in the `"deltas"` table and can henceforth be applied to work nodes.

The deltas implemented in the official API are:

- `infeasible_delta`,
- `bound_delta`,
- `semantics_delta`,
- `annotation_delta`,
- `table_delta` — this is converted to an `annotation_delta` before it is stored in the database,
- `point_delta` — a special `table_delta`,
- `boxes_delta` — a special `table_delta`,
- `dag_delta`,
- `split_delta`.

5.3. Small changes.

- The `c_write` program now uses the `c_write` report module instead of defining its own visitor.
- The `dag_simplify` program now uses the `dag_write` report module instead of being a standalone program.