AutoBayes
Program Synthesis System
System Internals

—Draft—

Johann Schumann, SGT, Inc.

NASA Ames Research Center
Draft — Internal Version: August 11, 2011

Preface

This document is a draft describing many important concepts and details of AutoBayes, which should be helpful in understanding the internals of AutoBayes and for extending the AutoBayes system. Details on installing AutoBayes, using AutoBayes, and many example specifications can be found in the AutoBayes manual[1].

This version of the document contains the supplemental information for the lecture on schema-based synthesis and AutoBayes, presented at the 2011 Summerschool on Program Synthesis (Dagstuhl, 2011).

This lecture combines the theoretical background of schema based program synthesis with the hands-on study of a powerful, open-source program synthesis system (AutoBayes).

Schema-based program synthesis is a popular approach toward program synthesis. The lecture will provide an introduction into this topic and discuss how this technology can be used to generate customized algorithms.

The synthesis of advanced numerical algorithms requires the availability of a powerful symbolic (algebra) system. Its task is to symbolically solve equations, simplify expressions, or to symbolically calculate derivatives (among others) such that the synthesized algorithms become as efficient as possible. We will discuss the use and importance of the symbolic system for synthesis.

Any synthesis system is a large and complex piece of code. In this lecture, we will study Autobayes in detail. AutoBayes has been developed at NASA Ames and has been made open source. It takes a compact statistical specification and generates a customized data analysis algorithm (in C/C++) from it. AutoBayes is written in SWI Prolog and many concepts from rewriting, logic, functional, and symbolic programming. We will discuss the system architecture, the schema libary and the extensive support infra-structure.

Practical hands-on experiments and exercises will enable the student to get insight into a realistic program synthesis system and provides knowledge to use, modify, and extend Autobayes.

[1] `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080042409_2008042209.pdf`

Contents

List of Figures

List of Tables

Listings

1. Starting AutoBayes

1.1 Command-line

Usually AutoBayes is called using the command line, e.g.,

```
autobayes -target matlab mix-gaussians.ab
```

where `mix-gaussians.ab` is the AutoBayes specification. Command line options and pragmas (see [?] and Appendix B) start with a"–".

1.2 Interactive Mode

Starting AutoBayes into the Prolog interactive mode can be done by

```
bash-3.2$ ../autobayes -interactive mog.ab

            +-------------------------------------------------------------+
            | AutoBayes V0.9.9    ------  Sat Jul  2 09:42:26 2011        |
            | Copyright (c) 1999-2011 United States Government            |
            | as represented by the Administrator of the National         |
            | Aeronautics and Space Administration.                       |
            | All Rights Reserved. Distributed under NOSA 1.3             |
            +-------------------------------------------------------------+

                        *** Interactive shell started ***

?- load('mog.ab').
Success  [mog.ab]: no errors found
true.

?- solve.
... << all logging messages >>
```

Listing 1.1: Starting AutoBayes into interactive mode

There is a number of commands available in the interactive mode of AutoBayes. These are defined in `interface/commands.pl`.

`load(+File)` loads specification file and constructs the AutoBayes model.

`clear` deletes the current AutoBayes model

`show` lists the current model.

`save(+File)` saves the current model in the AutoBayes specification syntax into a named file (unsupported).

`solve` attempts to solve the model and generate intermediate code, and list it on the screen.

This command also should place the generated code into the Prolog data base.

`????` command for generating code.

1.3 Loading AutoBayes into Prolog

```
1 $pl
2 % library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 2,284 bytes
3 Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.2)
4 Copyright (c) 1990-2010 University of Amsterdam, VU Amsterdam
5 SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
6 and you are welcome to redistribute it under certain conditions.
7 Please visit http://www.swi-prolog.org for details.
8
9 For help, use ?- help(Topic). or ?- apropos(Word).
10
11 ?- [main_autobayes].
12 <<lots of messages>>
13 ?-
```

Listing 1.2: Loading AutoBayes into Prolog

Note that here only the AutoBayes program code will be loaded but not any specification or command line flags/pragmas.

2. AutoBayes Architecture

2.1 Top-level Architecture

The top-level system architecture is shown in Figure 2.1

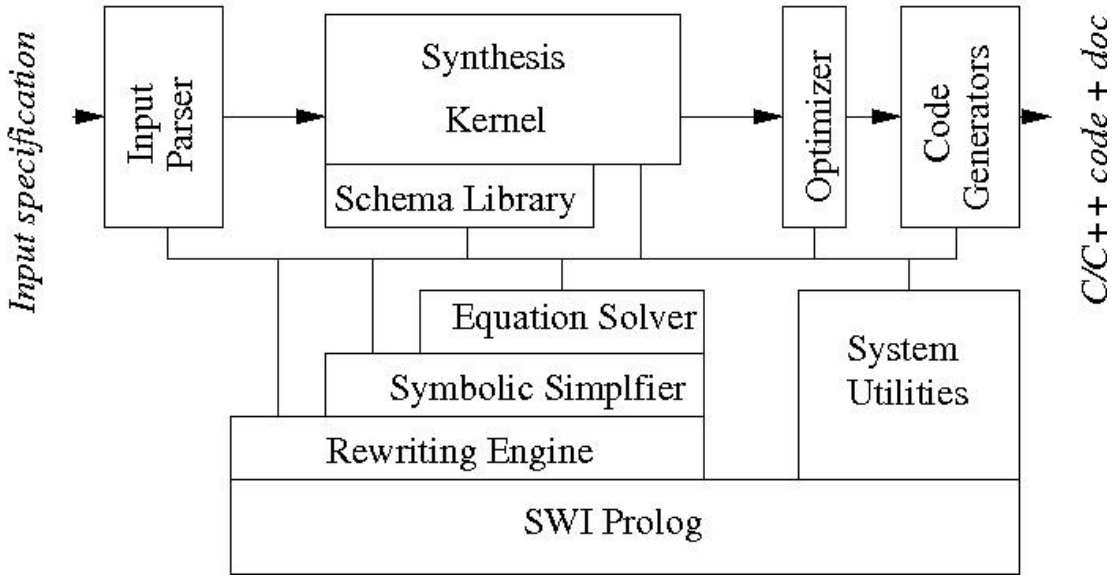

Figure 2.1: AutoBayes architecture

2.2 Directory Structure

The AutoBayes directory structure is shown in Figure 2.2. Note that a global shell variable, `AUTOBAYESHOME` must point to the top-level directory.

2.3 Synthesis and Code Generation

The synthesis and code generation parts are strictly separated. The synthesis kernel generates one or more customized algorithms and places them (using assert) into the PROLOG data base under the predicate name `synth_code(Stage, Code)`. After the

synthesis phase, the generated algorithms are retrieved one-by-one and code is generated for them. This is done by the predicate `main_cg_loop` (file: `toplevel/main.pl`).

Note that the synthesis component does not use any information about the code generation target.

Each subcomponent of AutoBayes can run individually. AutoBayes can dump the generated algorithm (`-dump synt`) into a file, which then can be read in by the AutoBayes codegenerator `main_codegen(DumpFile)`. This is accomplished with the command-line switch `-codegen`.

2.3.1 Synthesis

After opening and preprocessing the AutoBayes specification file (using the CPP preprocessor), the specification is read in using the prolog parser. Predicates for handling the specification are in `interface/syntax.pl`. All information is stored in the Prolog data base as the AutoBayes model. The goal statement

```
max pr(...)  for VAR_SET
```

actually triggers the program synthesis. It puts the information into the model as `optimize_target(...)`.

After reading the specification and processing the command line, the predicate

```
main_synth(+Specfile)
```

triggers the synthesis:

- the specification files is preprocessed
- all log-files are opened
- depending on the number of requested programs, the predicate `main_synth_loop` is calles, which calls the schema-based process `synth_arch/3`. If more than one program is requested, this predicate is visited again using backtracking. The program, which is generated during each call of that predicate is stored in the Prolog data base (non-backtrackable).
- After all requested programs have been generated, the synthesis part is finished. The actual generation of code is done using the predicate `main_cg_loop` (see below).

2.3.2 Code Generation

The code generator is parameterized by the target flag `-target`, which selects the code generation target as well as a number of pragmas.

The code generation is performed in several stages; stages, which are language-specific (e.g., C, C++, Ada) are marked with "L", those, which are target-specific with "T".

1. top-level: `main_cg_loop`
2. for each generated program in `synth_code(_,_)` perform the proper code generation
3. `main_cg_prog` performs:

   get name of generated program

   get and simplify complexity bound (if applicable)

   add declarations for the variables in the for-loops `loopvars`

   optimize the pseudo-code (`pseudo_optimize`)

   check for syntactic correctness of the intermediate code `pseudo_check`

   list the code after optimization `main_list_code('iopt',...`

   generate the actual code `cg_codegen(Code)`

The predicates for the actual code generation is in the directory `codegen` and subdirectories thereof. It's top-level predicate `cg_codegen(Code)` performs the following steps

1. open the symbol table
2. add (external) declarations
3. preprocess the code `cg_preprocess_code`

   get target language and target system

   preprocess the pseudo code `cg_preprocess_ps` (L,T)

   transform the code into language/target specific constructs `cg_transform_code` (L,T)
4. produce the code `cg_produce_code`

   open all files

generate headers `cg_generate_header`

generate include statements `cg_generate_includes`

generate declarations of global variables `cg_generate_globals`

produce code for each component (or for the main procedure). This is done using `cg_produce_component`, which then executes `cg_preamble`, `cg_generate_code`, and `cg_postamble`

produce end of HTML headers (why?)

close the files

5. list the code in various formats
6. produce the design document if desired

The `cg_preamble` is just a switchboard, which causes the generation of the interface code for the given procedure, the usage statement, and the input/output declarations. Similarly, the `cg_postamble` produces code at the end of the given procedure (e.g., handling of return values).

The switchboard for the code generation `cg_generate_code` is in the file `codegen/cg_code.pl` and finally calls `cg_generate_lowlevel_code`, which is specific for each target system and prints each statement one after the other.

2.3.3 Target Specific Code Generation

Lang	Target	cmdline-flags
C [c_c++]	Matlab	-target matlab
C [c_c++]	stand-alone	-target standalone
C++ [c_c++]	Octave	-target octave
ADA	stand-alone	-target ada

Table 2.1: Code generator options

Note that the ADA version is not fully supported.

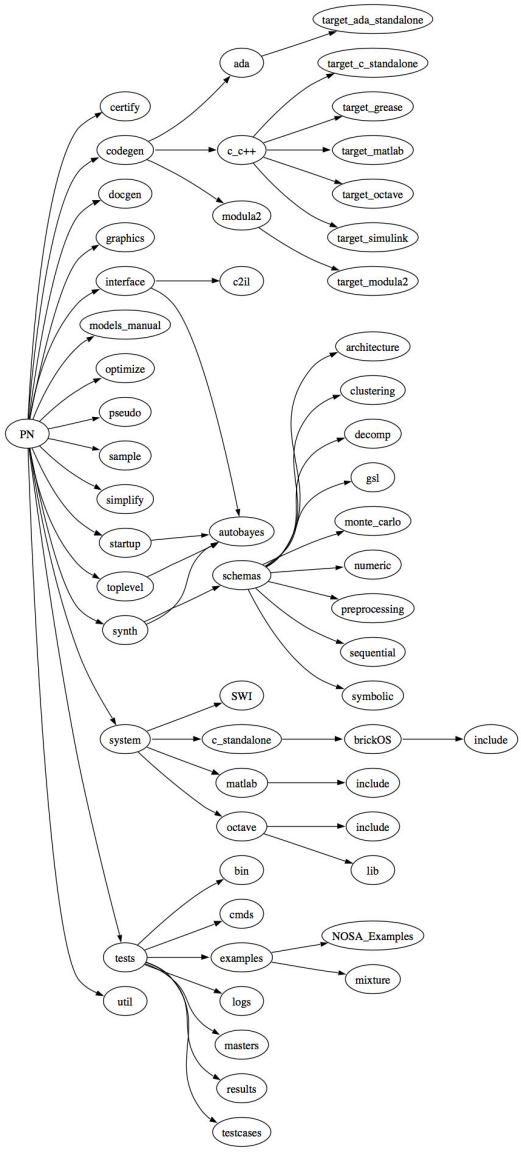

Figure 2.2: The directory structure of AutoBayes

3. The Schema System

The schema-based synthesis process is triggered by the goal expression in the AutoBayes specification. There are two different kinds of goal expressions

```
1 double mu.
2 data double x(1..10).
3 max pr{x|mu} for {mu}.
```

Listing 3.1: AutoBayes specification for a probabilistic optimization problem

```
1 double x.
2 max -x**2 + 5*x -7  for {x}.
```

Listing 3.2: AutoBayes specification for a functional optimization problem

Whereas the first form performs a probabilistic maximization (and triggers calls to `synth_schema`, the second form is a functional optimization and triggers `synth_formula_try`.

Note that all sorts of probability expressions in the goal are automatically converted into a `log_prob(...)` expression.

3.1 The synth_schema Predicate

The top-level schema predicate is

`synth_schema(+Goal, +Given, +Problem, -Program)`

In most cases, a 5-ary predicate is used to solve `log_prob(U,V)` problems:

`synth_schema(+Theta, +Expected, +U, +V, -Program)`

Note that the AutoBayes model has to be considered as an additional "invisible" argument. For efficiency reasons, AutoBayes implements the model using global (backtrackable) data structures. Otherwise, the model would need to be carried as additional arguments, like

`synth_schema1(+Theta, +Expected, +U, +V, +ModelIn, -ModelOut, -Program)`

That modification would not only be required for the top-level schemas, but also for all support predicates, making this approach cumbersome.

3.2 The synth_formula Predicate

This predicate is used to solve functional optimization problems. These schemas can be either called from the top-level or by other schemas in case some functional subproblem must be solved. The top-level predicate is:

`synth_formula(+Vars, +Formula, +Constraints, -Program)`

This predicate tries generate a program (or solve symbolically) that finds the optimum (maximum) values of the variables `Vars` in the formula `Formula` under the given constraints.

Note: `synth_formula_try` is a guarded front-end to `synth_formula` that handles stack and tracing. In particular, in case of failure, the dependencies must be restored using `depends_restore`.

Figure 3.1 shows the entire (static) schema hierarchy. Note, that during synthesis, one schema can trigger arbitrary other schemas in order to solve a given problem.

3.3 AutoBayes Schema Hierarchy

AutoBayes has a separate schema hierarchy for probabilistic and functional problems.

All schemas are in Prolog files, which are included in the file `synth/synth.pl`. Note that the order is important, as the schema-search uses Prolog's backtracking search.

```
1 :- discontiguous synth_schema/5.
2 :- multifile synth_schema/5.
3 :- discontiguous synth_schema/4.
4 :- multifile synth_schema/4.
5
6 :- discontiguous synth_formula/4.
7 :- multifile synth_formula/4.
8 :- dynamic synth_formula/4.
9 ...
10 synth_schema([], _, _, skip) :-
11         !.
12
13 :- ['schemas/preprocessing/scaling.pl'].
14
15 synth_schema(Goal, Given, log_prob(U,V), Program) :- ...
16
17 :- ['schemas/decomp/d_prob.pl'].
18 ...
```

Listing 3.3: AutoBayes Schema hierarchy — inclusion mechanism

3.3.1 AutoBayes probabilistic Schema Hierarchy

```
1 synth_schema([], _, _, skip) :- !.
2
3 :- ['schemas/preprocessing/scaling.pl'].
4 synth_schema(Goal, Given, log_prob(U,V), Program) :- ...
5
6 :- ['schemas/decomp/d_prob.pl'].
7 :- ['schemas/sequential/kalman.pl'].
8 :- ['schemas/sequential/sequent.pl'].
9 :- ['schemas/clustering/rndproject.pl'].
10 :- ['schemas/clustering/em.pl'].
11 :- ['schemas/clustering/kmeans.pl'].
12 synth_schema(Theta, Expected, U, V, Program) :-
13    % CONVERT PROBLEM TO PROBLEM OVER FORMULA FOR SYNTH_FORMULA/4
```

Listing 3.4: AutoBayes Schema hierarchy — probabilistic schemas

3.3.2 AutoBayes functional Schema Hierarchy

```
1 :- ['schemas/decomp/d_formula.pl'].
2 :- ['schemas/symbolic/lagrange.pl'].
3 :- ['schemas/symbolic/solve.pl'].
4 :- ['schemas/gsl/gsl-maximization.pl'].
5 :- ['schemas/numeric/section.pl'].
6 :- ['schemas/numeric/simplex.pl'].
7 :- ['schemas/numeric/generic.pl'].
```

Listing 3.5: AutoBayes schema hierarchy — functional schemas

3.3.3 AutoBayes Support Schema Hierarchy

Several schemas call special-purpose sub-schemas, e.g., to produce code for initialization. These predicates have non-standarized arguments and form individual hierarchies. An example are the schemas for producing initialization code for the clustering algorithms in `synth/schemas/clustering/clusterinit.pl` with the main schema predicate

```
ci_center_select(+CenterName, +DataIn, +IPointsIn, +IClassesIn, +CDim, -Program)
```

3.4 Adding a New Schema

3.4.1 Example 1

This example is an existing schema in AutoBayes, which, given a log_prob problem, tries to solve it symbolically or as a numerical optimization problem. This schema can be found in `synth/synth.pl` and has been abbreviated. In particular, all generation of explanations have been removed for clarity.

```
1  synth_schema(Theta, Expected, U, V, Program) :-
2      % Decompose as far as possible
3          cpt_theorem(U, V, Prob, rels(Theta)),
4
5      % Check whether Prob is atomic and if so, replace it by the
           density
6    prob_replace(Prob, Prob_formula),
7
8       % extract model constraints
9    model_constraint(Pre_constraint),
10   simplify(Pre_constraint, Constraint),
11
12   copy_term(Expected, Expected_copy),
13   synth_sum_expected(Expected_copy, log(Prob_formula), Pre_formula),
14   pv_lift_existential(Pre_formula),
15   simplify(Constraint, Pre_formula, Formula),
16   assert_trace(trace_schema_inout, 'synth/synth.pl',
17           ['Log-Likelihood function:\n', Formula]),
18
19     % Build the dependency graph from the simplified formula,
20     % recurse on the Formula and clean up.
21   depends_save,
22   depends_clear,
23   depends_build_from_term(Formula),
24
25     % Find closed-form solution
26   synth_formula_try(Theta, Formula, Constraint, Step),
27
28     % Clear stack
29   depends_restore,
30
31      % compose the program
32   Program = series([Step],
33     [comment('lots of text ...'),f_loglikelihood(Formula)]).
```

Listing 3.6: Example Schema

The schema in Listing 3.6 is called with the statistical variables `Theta` and the expected variables `Expected`, as well as `U, V`, which are the arguments of the log_prob problem.

The first two subgoals decompose the problem statistically (using the AutoBayes model) and, if successful return the probability `Prob` to be solved. Then it is checked if this probability is atomic, i.e., it is not conditional. The resulting formula `Prob_formula` must be considered. This predicate also replaces all PDFs (e.g., `gauss`) by the corresponding symbolic formulas (see Chapter 4)). These two predicates comprise the guards for this schema. In order to obtain the (numerical) formula that is to be optimized, the following steps must be carried out.

In parenthesis are the values for the normal-example.

The predicate is called with `synth_schema([mu, sigma], [], [x(_)], [mu,sigma], Program)`.

The probability formula is

$$\prod_{i=0}^{-1+n} \mathbf{P}(x_i|\mu,\sigma^2)$$

With the PDF replaced, the problem to solve (`Pre_formula`) becomes

$$\log \prod_{i=0}^{-1+n} \exp\left(\frac{-\frac{1}{2}\left(x_i-\mu\right)^2}{\left(\sigma^2\right)^{\frac{1}{2}2}}\right) \frac{1}{\sqrt{2\,\pi}\left(\sigma^2\right)^{\frac{1}{2}}}$$

where the constraints, coming from the model are

```
and([and([not(0=n_points), 0=<n_points]),
     and([not(0=sigma_sq),0=<sigma_sq]),
    type(mu,double),type(n_points,nat),type(sigma_sq,double),type(x,double)])
```

- because the log-likelihood is maximized, a logarithm of the probabilistic formula must be taken.
- This formula is the transformed into a sum with respect to the `Expected` values.
- This sum is simplified under the given constraints
- the schema-driven solution of the problem is tried `synth_formula_try` and a code segment is returned in `Step`.

- The final program segment is a code block containing that code segment
- After processing, dependencies must be restored.

3.4.2 Example 2

3.5 Notes

- scaling: must extract sigma
- loop around EM: flag controlled or statistics controlled
- numerical optimization: regula falsi
- multivariate optimization full synthesis, based upon gsl utilities

3.6 Schema Control

Prolog backtracking search

multiple programs (-maxprog)

multiple programs with complexity (unsupported)

control via pragmas

schema-control language

Figure 3.1: The static schema-hierarchy for AutoBayes

4. Probability Density Functions

Probability Density Functions (PDFs) and their properties can be defined easily. In order to add a new PDF, e.g., `mypdf`, two places must be modified: (1) the symbol must be made a special symbol for the input parser (file `interface/symbols.pl`) and (2) define the properties in the file `synth/distribution.pl`.

As an example, the PDF `mypdf` should have the same properties as the regular Gaussian, i.e., be defined for one variable and should have 2 parameters, e.g., $X \sim$ `mypdf`(a, b).

```
1 symbol_distribution(mypdf, 1, 2).
```

Listing 4.1: Definition of PDF symbol in `interface/symbols.pl`

```
1  dist_density(X, mypdf, [A, B],
2               (1/(sqrt(2*pi) * B)) * exp((-1/2) * ((X - A)**2) / B**2)
3              ) :-
4          !.
5
6  dist_mean(mypdf, [A, _], A) :-
7          !.
8
9  dist_mode(mypdf, [A, _], A) :-
10         !.
11
12 dist_variance(mypdf, [_, B], B**2) :-
13         !.
14
15 dist_constraint(mypdf, [_, B],
16                  not(0 = B)
17                 ) :-
18         !.
```

Listing 4.2: Definition of PDF `synth/distribution.pl`

For each PDF, its density with respect to the parameters must be given, the mean, the mode, and the variance. Specific constraints for each PDF can be given. However, the current version of AutoBayes does not use these constraints.

4.1 The AutoBayes Model

The statistical model, as given by the specification is stored in a global data structure, the *model*. The predicates concerning handling of the model are mainly in the file `synth/model.pl`.

4.1.1 The Model Data Structure

The current contents of the entire model can be printed or written into a stream using the predicate `model_display`. The predicate names (e.g., `model_name`) are those that are stored in the prolog data base in a backtrackable manner using `bassert` and `bretract`.

```
?- model_display.

Model: mog

Vers.: 0

%================ Names:
model_name(x)
model_name(c)
...
%================ Types:
model_type(x,double)
model_type(c,nat)
model_type(sigma,double)
...
%================ Constants:
model_constant(n_classes)
model_constant(n_points)
%================ Outputs:
model_output(c)
%================ Variables:
model_var(x)
model_var(c)
model_var(sigma)
model_var(mu)
model_var(phi)
%================ Random:
model_random(x)
model_random(c)
...
%================ Indexed:
model_indexed(x,[dim(0,+[-1,n_points])])
model_indexed(c,[dim(0,+[-1,n_points])])
...
```

```
35 %================ Distributions:
36 var_distributed(x(A),gauss,[mu(c(A)),sigma(c(A))])
37 var_distributed(c(A),discrete,[vector((B:=0.. +[-1,n_classes]),phi(B))
       ])
38 %================ Knowns:
39 var_known(x(A))
40 %================ Constraints:
41 var_constraint(sigma(A),and([not(0=sigma(A)),0=<sigma(A)]))
42 var_constraint(phi(A),0= +[-1,sum([idx(B,0,+[-1,n_classes])],phi(B))])
43 var_constraint(n_points,n_classes<<n_points)
44 ...
45 %================ Optimize:
46 optimize_target([mu(A),phi(B),sigma(C)],[],log_prob([x(D)],[mu(E),phi(
       F),sigma(G)]))
```

Listing 4.3: Displaying the AutoBayes model

4.1.2 The Model Stack

During the synthesis process, schemas can modify the model. Since the schema-based synthesis process is done using a search with backtracking, changes to the model must be un-done in case a schema fails.

Therefore, AutoBayes uses a backtrackable data structure for the model and a model stack. Before a schema or subschema modifies a model, it usually generates a copy of the model (`model_save`) on the stack. That copy then can be modified, destroyed, or the old model restored with a pop on the stack `model_restore`).

The individual predicates are:

`model_clear/0` remove any modifiable model parts

`model_destroy/0` completely remove a model from the database

`model_save/0` save modifiable model parts at next level

`model_restore/0` restore modifiable model parts to previous level

TODO: FIGURE on SCHEMA hierarchy and model stack

4.1.3 Modifying the Model

The model can be modified using predicates in `synth/model.pl`. E.g., `model_makeknown(X)` makes the statistical variable `X` known in the model.

4.2 Statistical (Bayesian) Decomposition

TODO

5. Low-level Components of AutoBayes

5.1 Command Line Options and Pragmas

AutoBayes is called from the command line with command line options and pragmas. Command-line options (starting with a "–") control the major operations of AutoBayes. Pragmas are a flexible mechanism for various purposes, like setting specific output options, controlling individual schemas, or for debugging and experimentation.

5.1.1 Pragmas

In AutoBayes all pragmas are implemented as Prolog flags. The command-line interpreter analyzes all tokens starting with `-pragma` and sets the flag accordingly.

Pragmas can be set inside an AutoBayes specification using the flag directive, e.g.,

:– flag(schema_control_init_values, _, automatic).

There, no check of validity of the flag's name or its value is performed.

Adding a new Pragma

All pragmas are defined in the file `startup/flags.pl`. Pragmas are declared by a `pragma/6` multifile predicate:

```
pragma(SYSTEM, NAME, TYPE, INIT, VL, DESC).
```

where

SYSTEM	=	_ — 'AutoBayes' — 'AutoFilter'
NAME	=	name of pragma = name of flag
TYPE	=	boolean — integer — atomic — callable ...
INIT	=	initial value
VL	=	[[V,E], ...] possible values and explanations
DESC	=	atom containing description

```
1 pragma('AutoBayes', schema_control_init_values, atomic, automatic,
2         [
3           [ automatic , 'calculate best values'],
4           [ arbitrary , 'use arbitrary values'],
```

```
5           [ user , 'user_provides_values_(additional_input_parameters'
                ]
6         ],
7         'initialization_of_goal_variables').
8
9 pragma('AutoBayes', schema_control_solve_partial, boolean, true, [],
10        'enable_partial_symbolic_solutions').
11
12 pragma('AutoBayes', example_pragma, integer, 99, [],
13        'Example_for_an_integer_pragma').
```

Listing 5.1: AutoBayes pragmas

5.2 Backtrackable Global Data

The schema-based synthesis process of AutoBayes uses PROLOG's backtracking mechanism. In particular, the statistical model is modified during the search process by schemas. These changes must be undone during backtracking.

Since the AutoBayes model is kept as a global data structure in the Prolog data base, mechanisms for backtrackable global data structures, namely flags and counters had to be developed.

These predicates have been implemented in C as external predicates.

NOTE: More receent versions of SWI Prolog might have similar mechanisms already incorporated.

5.2.1 Backtrackable Flags

Backtrackable flags are indexed by an natural number between 0 and N, where N is fized during compile time (`system/SWI/bflag.c`).

The predicate `pl_bflag(+N, -V1, +V2)` gets the current value of backtrackable flag number N in V1 and sets a new value in V2. Getting and setting values are done in the same way as for the standard Prolog `flag/3`.

5.2.2 Backtrackable Counters

Similar to AutoBayes counters `util/counter.pl`, backtrackable counters are defined by the following predicates

```
1 bcntr_new(C) :- % NEW COUNTER
2 bcntr_set(C,M) :- % SET THE COUNTER
3         % THE PREDICATES BELOW ARE BACKTRACKABLE
```

```
4 bcntr_get(C,N) :- % GET THE CURRENT COUNTER VALUE
5 bcntr_inc(C) :- % INCREMENT THE COUNTER
6 bcntr_inc(C, Incr) :- % INCREMENT COUNTER BY Incr
7 bcntr_dec(C) :- % DECREMENT THE COUNTER
```

Listing 5.2: Interface predicates for backtrackable counters

5.2.3 Backtrackable Bitsets

Backtrackable bitsets have been implemented as external predicates in C to enable backtrackable asserts and retracts. The extension defines one global backtrackable bit set for integers `1..BSET_DEFAULT_LENGTH` and two interface predicates: `inbset(X)` succeeds if number `X` is in the bit set, `setbset(X,1)` adds number `X` to the bit set, and `setbset(X,0)` removes number `X` from the bit set. The latter two predicates are backtrackable. Note that bitsets are used only for the implementation of backtrackable asserts/retracts (see Section 5.2.4).

5.2.4 Backtrackable Asserts/Retracts

This module contains the Prolog-support for backtrackable asserts and retracts, i.e., an assert/retract-mechanism which is integrated with the normal backtracking mechanism of Prolog. An N-ary predicate F is declared as a backtrable predicate via

```
:- backtrackble p/1.
```

in a similar way to a `dynamic`-declaration. Backtrable asserts and retracts are done via bassert and beretract. Here, "backtrackable" means that the assertions are undone on backtracking by the Prolog-engine the same way variable bindings are undone, e.g.:

```
1 q(X) :-
2         ...,
3         bassert(p(a)),  %%% WILL BE UNDONE/RETRACTED ON BACKTRACKING
4         ...,
5         fail,
6         ...,
7 q(X) :-
8         ...,
9         p(a),           %%% FAILS
10        ...,
```

Listing 5.3: Backtrackable assertions

5.3 Data Structures and Their Predicates

see files and their documentation in `util`

bag.pl Predicates for a Prolog representation of bags

diffset.pl Predicates for a compact representation of differences between arbitary term sets (see termset.pl)

equiv.pl calculates the equivalence class of a binary relation that is given as a list of lists

listutils.pl Predicates for handling of lists

meta.pl Meta-operations on uninterpreted Prolog terms (e.g., unification, etc.)

stack.pl Prolog representation of a stack

subsumes.pl subsumption check

term.pl Prolog representation for AC terms

termset.pl Prolog representation for sets of terminstances.

topsort.pl topological sort

trans.pl calculates the transitive closure of a relation

5.4 The Rewriting Engine

A rewriting engine has been implemented on top of Prolog. Rewriting rules are given a Prolog clauses, which are being compiled for efficieny reasons.

5.4.1 Rewriting Rules

The rules for rewriting must be given as a predicate of the form

```
rule(+Name, +Strategy, +Prover, +Assumptions, +TermIn, ?TermOut)
```

where the parameters have the following meaning:

Name string or atom used to identify the rewrite rule (e.g., in tracing); should be unique.

Strategy a strategy vector of the form

```
[eval=Evaluation, flatten=Bool, order=Bool, cont=Continuation]
```

associated with each rule. Evaluation must be either eager or lazy; Continuation is either a Bool or a rule name. Rules with strategy `[eval=eager|_]` are applied a first time in a top-down fashion (i.e., before the subtrees are normalized). Rules with strategy `[eval=lazy|_]` are applied in a bottom-up fashion. If the continuation-argument of `rwr_cond` is fail, pure bottom-up rewriting is implemented, otherwise dovetailing is implemented (i.e., exhaustive rewriting). Use the strategy vector `[eval=lazy|_]` as default for all rules

Use the strategy vector `[eval=lazy|_]` as default for all rules to get the complete innermost/outermost strategy. Use a rule

`rule('block-f', [eval=eager,_,_,cont=fail|_], _, _, f(X), f(X)).` to prevent rewriting from all subtrees with root symbol f.

Prover currently not used

Assumptions Use the assumption 'true' for unconditional rewriting

TermIn Term to be normalized.

TermOut Result of rule application.

Simple rewriting rules are just unit clauses or complex rules with bodies (Listing 5.4).

```
1  expr_optimize(_, 'expr-reintroduce-reciprocal',
2                [eval=lazy|_],
3                _,_,
4                Term ** (-1),
5                1 / Term
6               ) :-
7          !.
8
9  expr_optimize(Level, 'expr-reintroduce-subtraction',
10               [eval=lazy|_],
11               _,_,
12               +(Summands),
13               Subtraction
14              ) :-
15         Level > 0,
16         list_split_with(factors_negate, Summands, Neg, Pos),
17         Neg \== [],
18         !,
19         (Pos cases [
20            []  -> expr_mk_subtraction(Neg, Subtraction),
21            [P] -> expr_mk_subtraction(P, Neg, Subtraction),
22            _   -> expr_mk_subtraction(+(Pos), Neg, Subtraction)
23          ]
24         ).
```

Listing 5.4: Examples for Rewriting Rules

5.4.2 Compilation of Rewriting Rules

A (customized) set of rewriting rules is compiled into a ruleset using the directive `rwr_compile`. Note that the individual groups of rewriting rules can be placed in separate files.

```
1  ruleA('ruleA:1', [eval=eager|_], _, _,
2            Source, Target) :- !.
3  ruleA('ruleA:2', [eval=eager|_], _, _,
4            Source, Target2) :- !.
5
6  ruleB('ruleB:1', [eval=lazy|_], _, _,
7            Source, Target) :- !.
8           ...
9
10
11 :- rwr_compile(myruleset,
12           [
13           rulesA,
14           rulesB,
15           ...
16           ]).
17
18 do_rewrite(S, T) :-
19           rwr_cond(myruleset, true, S, T).
20
21 do_rewrite_timelimit(S, T, Max) :-
22           call_with_time_limit(Max, rwr_cond(myruleset, true, S, T)).
23 do_rewrite_timelimit(S, S, _).
```

Listing 5.5: Compilation of Rewriting Rules and top-level calls

6. The Symbolic System

AutoBayes uses its symbolic subsystem extensively. The system is in part implemented as rewriting rules and in part as Prolog predicates.

6.1 Top-Level Predicates

Some of the common top-level predicates are

`simplify(S, T)` simplifies expression S and returns T

`simplify(Assumptions, S, T)` simplifies expression S and returns T under the given assumptions.

`range_abstraction(+S, -Range)` provides a range abstraction for S.

`range_abstraction(+Assumptions, +S, -Range)` provides a range abstraction for S under the given assumptions.

`defined(S, Condition)` provides a definedness constraints for S.

`defined(Assumptions, S, Condition)` provides a definedness definition for S under given constraints.

`solve(Assumptions, Var, Equation, Solution)` calls the symbolic equation solver to solve the equation `Equation` for the variable `Var` under the given assumptions.

`leqs_solve(Assumptions, Vars, Equations, Solution)` attempts to solve symbolically a system of linear equations and returns a solution, using a Gaussian elimination. This predicate can use local program variables for sub expression, so a `let(...)` expression is returned.

Note that for this predicate, the terms must be in list-notation.

```
1 ?- simplify((a+b)*(a-b),T),print_expr(user_output,0,T,_).
2 -1 * b ** 2 + a ** 2
3 T = +[*([-1, b**2]), a**2] .
4
5 ?- simplify(sin(x)**2 + cos(x)**2,T).
6 T = 1 .
7
8 ?- defined(1/x,D).
```

```
 9 D = not(0=x) .
10
11 ?- defined(tan(x),D).
12 D = not(0=cos(x)) .
13
14 ?- solve(true, x, 5*x**2 - 3 = 0, S), print_expr(user_output,0, S, _)
       .
15 1 / 10 * 60 ** (1 / 2)
16 S = *([1/10, 60** (1/2)]).
17
18 ?- solve(true, x, 17*x - 3 = 0, S), print_expr(user_output,0, S, _).
19 3 / 17
20 S = 3/17.
21
22 ?- leqs_solve([],[x,y],[x,*([5,y])],Y).
23 Y = let(local([]), series([skip, skip, skip, skip, skip, skip, skip],
       []), [y=0, x=0]) .
24
25 ?- leqs_solve([],[x,y],[x,+([5,y])],Y).
26 Y = let(local([]), series([skip, skip, skip, skip, skip, skip, skip],
       []), [y= -5, x=0]) .
27
28 ?- leqs_solve([],[x,y],[x,+([5,y,x])],Y).
29 Y = let(local([]), series([skip, skip, skip, skip, skip, skip, skip],
       []), [y= -5, x=0]) .
30
31 ?- leqs_solve([],[x,y],[+([x,1]),+([5,y,x])],Y).
32 Y = let(local([]), series([skip, skip, skip, skip, skip, skip, skip],
       []), [y= -4, x= -1])
```

Listing 6.1: Examples for symbolic subsystem

6.2 Program Variables

The AutoBayes system distinguishes between different kinds of variables. This is necessary, because there are Prolog variables, which have to be distinct from code variables, which show up in the generated code fragments. The latter type of variable is called *program variable*.

Program variables are not represented by Prolog variables (because no unification can be allowed there), but by a reserved term `pv(n)`, where n is a number. Such program variables can be universally quantified or existentially quantified. The latter is used, e.g., to convert Prolog variables in a term into actual variable names.

During pretty-printing or in the final code, existential variables are printed as `pv###`,

e.g., pv96.

```
1 % get a new fresh (extensial variable). The "pv1" is the
2 % external format
3 ?- pv_fresh_existential(X), print_expr(user_output,0,X,_).
4 pv1
5 X = pvar(1).
6
7 % convert index variable for a sum into program variables
8 ?- C=sum(idx(X,0,10),d(X)),
9     pv_lift_existential(C),
10    print_expr(user_output,0,C,_).
11 sum(pv3 := 0 .. 10, d(pv3))
12 C = sum(idx(pvar(3), 0, 10), d(pvar(3))),
13 X = pvar(3).
```

Listing 6.2: Predicates for program variables

7. Pretty Printing and Text Generation

7.1 Pretty Printer

A piece of pseudo-code can be pretty-printed using `pp_pseudo(+Stmt)`. It pretty-prints the statement onto the screen (or into a file if a stream is given as the first argument).

An expression can be printed into a stream using `print_expr(+Stream, +Indent, +Expr, ?NewPos)`.

The syntax definition of the intermediate language is given in Appendix A.

```
1 ?- pp_pseudo(assign(x,5*x**3 -5,[comment('initial value')])).
2 // initial value
3 x := 5 * x ** 3 - 5;
4 true .
5
6 ?- print_expr(user_output, 0, x**2+cos(x), _).
7 x ** 2 + cos(x)
8 true.
```

Listing 7.1: Printing statements and terms

7.2 Pretty Printer for LaTeX and HTML

Generating an HTML or LaTeX representation of an expression or a piece of code, the same pretty-printer interface is used. The actual output format is controlled by various flags.

`pp_latex_output` if set to 1, LaTeX output will be generated

`pp_html_output` if set to 1, HTML output will be generated

Additional predicates in `pp_*.pl` provide support to writing headers, etc.

```
1 ?- pp_pseudo(assign(x,x+1,[comment('update x')])).
2 // update x
3 x := x + 1;
4 true
5
```

```
?- flag(pp_latex_output,_,1),
   pp_pseudo(assign(x,x+1,[comment('update x')])).

/*@\setlength{\mywidth}{0pt}\addtolength{\mywidth}
{78\myspace}\begin{minipage}{\mywidth}\small\vspace*{0.5ex}
\rm\em\noindent{}update x\end{minipage}@*/
x := x + 1;
true .

?- flag(pp_latex_output,_,0),
   flag(pp_html_output,_,1),
   pp_pseudo(assign(x,x+1,[comment('update x')])).
<font color="green">
// update x<br></font>
x := x + 1;<br></tt>
</body>
</html>
true
```

Listing 7.2: Pretty printing to LaTeX and HTML

7.3 Support for Text Generation

Generation of explanations and comments in the synthesized code is of great importance. Only a well-documented autogenerated algorithm can be used and understood. AutoBayes contains a number of predicates to facilitate the generation of text fragements to explain schemas and code. These texts are handled as comments in the intermediate language and stored as `comment(...)` in the attribute list, e.g., `assign(x,0,[comment('Initial assignment')])`.

The full powered schema-based synthesis approach requires that the explanation text can be customized accordingly for scalars, vectors, matrices; single elements and enumeration lists, etc. Predicates in `synth/lexicon.pl` provide functionality for this purpose.

```
lex_probability_atom(Prob, XP_prob),
(XP_prob = *(Prob_args)
 -> true
  ; Prob_args = []
),
lex_numerus_align('The ',        Prob_args, XP_prob_article),
lex_numerus_align('probability', Prob_args, XP_prob_numerus),
lex_numerus_align('is',          Prob_args, XP_prob_verb),
lex_numerus_align('function',    Prob_args, XP_prob_density),
lex_enumerate_vars(Theta, XP_theta),
```

```
lex_probability_atom(Prob, XP_prob_atom),
(Expected = []
 -> XP_likelihood = [
       ' This yields the log-likelihood function', expr(Pre_formula),
       'which can be simplified to', expr(Formula)
    ]
  ; (maplist(arg(1), Expected, EVars_list),
     flatten(EVars_list, EVars),
     lex_enumerate_vars(EVars, XP_EVars),
     XP_likelihood = [
       ' Summing out the expected ', XP_EVars,
       ' yields the log-likelihood function', expr(Pre_formula),
       'which can be simplified to', expr(Formula)
     ]
    )
),
XP = [
  'The ', XP_p_type, XP_p, ' is under the dependencies given in the ',
  'model equivalent to ', expr(XP_prob_atom),
  XP_prob_article, XP_prob_numerus, ' occuring here ', XP_prob_verb,
  ' atomic and can thus be replaced by the respective probability ',
  'density ', XP_prob_density, ' given in the model.', XP_likelihood,
  'This function is then optimized w.r.t. the goal ', XP_theta, '.'
],
```

Listing 7.3: Generation of Explanation in a schema `synth/synth.pl`

Appendix A. AutoBayes Intermediate Language

NOTE: The BNF description of the AutoBayes intermediate language is not up-to-date

AutoBayes uses a simple procedural intermediate language when it synthesizes code. This language is kept through all stages (synt, iopt, lang), until at the final stage, code in the target language's syntax is produced.

The intermediate code for AutoBayes is a relatively generic (procedural) pseudo code which contains specific means for handling numeric data and data structures like vectors and arrays. Syntactically, a program in that pseudo-code is a term as defined below.

For extended purposes, `ATTR` is introduced for most language constructions. They will contain attributes (e.g., state of initialization of the variable) or annotations which could contain explanations. `ATTR` is a list of well-formed (opaque) terms or the empty list `[]`.

A.1 Code

This top-level functor splits the program into a declarations and statements parts

```
PSEUDO_PROGRAM ::=
    prog ( IDENT, DECLS , STMT, ATTR )
```

Changes: code now contains a full list of declarations. `IDENT` will be the name of the function/program.

A.2 Declarations

All identifiers used within the code must have appropriate declarations; the only exceptions are index variables occurring within sums, loops, etc., as such constructs can easily be transformed into individual blocks containing the local declarations at the beginning of the construct. [1]

[1] Note that this requires different names for loop variables which occurr in nested loops.

Constants and variables are declared in a declaration block at the beginning of the program. The declaration block distinguishes between `constant` values, `input`, which are the parameters given to the synthesized routine, `output` which are the results returned by the synthesized routine, and `local` variables.

Symbolic model constants as for example the dimensions of vectors are represented either as `constants` if their value is given by the model or can be derived from other given constants or input variables or as `input` variables if their value must be supplied at runtime.

```
DECLS ::=
    decls(
          constant( [ DECL_LIST ] ),
          input( [ DECL_LIST ] ),
          output( [ DECL_LIST ] ),
          local( [ DECL_LIST ] )
         )

DECL_LIST ::=
    DECL
    | DECL , DECL_LIST

DECL ::=
    SCALAR_DECL
    | VECTOR_DECL
    | MATRIX_DECL
    | ARRAY_DECL


SCALAR_DECL ::=
    scalar( IDENT, TYPE_IDENT, ATTR )

VECTOR_DECL ::=
    vector( IDENT, TYPE_IDENT, [ DIM_LIST ], ATTR )

MATRIX_DECL ::=
    matrix( IDENT, TYPE_IDENT, [ DIM_LIST ], ATTR )

ARRAY_DECL ::=
    array( IDENT, TYPE_IDENT, [ DIM_LIST ], ATTR )
```

```
TYPE_IDENT ::=
double
| float
| int
| bool
```

Changes: declarations for vectors are similar to the old format, but now also contain the lower bounds. Note: giving the name with a set of FVARS only introduces IDENT/n not IDENT/0 and IDENT/n

variables marked `const` never occur on the left hand side of an assignment.

A.3 Indices and dimensions for vectors, arrays, and matrices

All indices into vectors or arrays (e.g., for declaration, iterative constructs) are given as lists of triples with the functor `idx`. For specification of vector/matrix/array dimensionality, the construct `dim(E1,E2)` is used, where the constant expressions `E1` and `E2` define the lower and upper bound of one dimension of the data object.

```
IDX_LIST ::=
    IDX
    | IDX , IDX_LIST

IDX ::=
idx( IDENT , EXPR , EXPR )

DIM_LIST ::=
    DIM
    | DIM , DIM_LIST

DIM ::=
dim( EXPR , EXPR )
```

The `IDENT` is the loop variable, the `EXPR`s are the lower bound and upper bound respectively.

A.4 Attributes

Attributes are opaque lists of terms used for various purposes, like attachments of comments or explanations or parameters (like target system, optimization level).

```
ATTR ::=
[]
| [ LIST_OF_ATTR ]

LIST_OF_ATTR ::=
    AT
    | AT , LIST_OF_ATTR
```

Example attributes which are currently being used are:

```
AT ::=
    file(IDENT)
    | target_language(LANGUAGE)
    | indent(NUMBER)
    | verbosity(NU)
    | linewidth(NU)
    | pedantic
    | target(TARGET)
    | comment(COMMENT)
    | initialize(EXPR)
    | ...

LANGUAGE ::=
    c | cplusplus
TARGET ::=
    matlab | octave
```

`file` The code-generation module will output the resulting code into the file `file`. This attribute is only evaluated on the top-level attribute-list of the `prog`.

`target_language` Select a target language for the code to be generated (overridden by selection of the target system). This attribute is only evaluated on the top-level attribute-list of the `prog`.

`indent` indentation level for formatting (default: 2). This attribute is only evaluated on the top-level attribute-list of the `prog`.

`linewidth` maximal length of a line in produced output code (default: 80). This attribute is only evaluated on the top-level attribute-list of the `prog`.

`verbosity` This is the verbosity level of the code-generation subsystem.

`initialize` This attribute is used for the declaration part only. A skalar variable is being initialized to the value given by `EXPR`. `EXPR` must be a simple expression (i.e., must not contain any pseudo-code instructions which evaluate into statements (like sum,norm,...).

`comment` Comments can be an atom or a list of atoms. Long lines are broken up into several shorter lines. Comments can have the following control atoms (must be present as single atoms):

`\ n` forces an immediate line-break

`labelref(label)` prints a reference to the label `label` defined elsewhere.

`label(label)` defines a label for later reference. In the current version, a label is printed as an additional comment.

In the current version, only the following attributes are evaluated for each statement: `comment, label`.

A.5 Statements STMT

```
STMT ::=
    SERIES
    | BLOCK
    | FOR_LOOP
    | IFSTAT
    | ASSIGN
    | WHILE
    | ASSERT
    | CALL_STAT
    | CONVERGING
    | ANNOTATION
    | FAIL
    | SKIP

STMT_LIST ::=
     STMT
    | STMT , STMT_LIST
```

A.5.1 fail and skip

`fail` generates a run-time error and/or exception and aborts processing of that function. `skip` just does nothing.

```
FAIL ::=
    fail(ATTR)

SKIP ::=
    skip(ATTR)
    | skip
```

A.5.2 Sequential Composition

```
SERIES ::=
    series ( [ STMT_LIST ] , ATTR )

BLOCK ::=
    block ( local([ DECL_LIST ]) , STMT , ATTR )
```

A.5.3 Annotations

```
ANNOTATION ::=
    annotation( TERM )
```

Annotations are placed "as is" into the code.

A.5.4 For-Loops

```
FOR_LOOP ::=
    for( [ IDX_LIST ], STAT, ATTR )
```

A.5.5 If-then-else

```
IFSTAT ::=
    if ( EXPR , STAT , STAT , ATTR )
```

A.5.6 While-Converging

```
CONVERGING ::=
    while_converging ( [ VECTORLIST ] , EXPR, STAT , ATTR)
```

Change: The EXPR evaluates to the tolerance down to which the iteration is to be performed.

```
VECTORLIST ::=
    VECTORDECL
    | VECTORDECL , VECTORLIST
```

A.5.7 While and Repeat Loop

```
WHILE ::=
    while ( EXPR , STAT, ATTR )

REPEAT ::=
    repeat ( EXPR , STAT, ATTR )
```

A.5.8 Assertion

```
ASSERT ::=
    assert( EXPR, TERM , ATTR)
```

Changes: This assert is to be used instead of the construct `if (expr,stat, fail)` The TERM is opaque and will be used in conjunction with explanation techniques.

A.5.9 Assignment Statement

```
ASSIGN ::=
    SIMPLE_ASSIGN
    | MULTIPLE_ASSIGN
    | SIMUL_ASSIGN
    | COMPOUND_ASSIGN

SIMPLE_ASSIGN ::=
    assign( LVALUE , EXPR , ATTR)

MULTIPLE_ASSIGN ::=
    assign_multiple( LVALUE_LIST, EXPR, ATTR )

SIMUL_ASSIGN ::=
    assign_simul( LVALUE_LIST, EXPR, ATTR )

COMPOUND_ASSIGN ::=
```

```
    assign_compound([IDX_LIST], LVALUE, EXPR, ATTR )
```

Note: the compound assignment will not be available in the current version.

```
LVALUE ::=
    VAR
    | VAR ( EXPR_LIST )
```

A value gets assigned to a skalar variable or an array access.

A.5.10 Misc. Statements

```
SOLVER_STAT ::=
    unsolved(LABEL , STAT )
    | poly_solver ( ...)
    | ...
```

A.6 Expression EXPR

```
EXP_LIST ::=
    EXPR
    | EXPR , EXPR_LIST

EXPR ::=
    NUMERIC_CONSTANT
    | CONSTANT
    | VAR
    | VAR ( EXPR_LIST )
    | - EXPR
    | PRE_OP
    | EXPR OP EXPR
    | SUM_EXPR
    | NORM_EXPR
    | MAXARG_EXPR
    | ( EXPR )
    | NUMFUNC
    | BOOLFUNC
    | CONDEXPR
    | attr( EXPR , ATTR )
```

The `attr` can be used to give attributes to atomic expressions and/or expressions without a leading function symbol.

```
NUMERIC_CONSTANT ::=
    0 | 1 | ..
    | FLOAT
    | pi

CONSTANT ::=
    identifier

VAR ::=
    identifier

OP ::=
    + | - | * | ** | /

PRE_OP ::=
    sdiv(EXPR, EXPR)
    | ssqrt(EXPR)
    | slog(EXPR)
```

The operators `sdiv, ssqrt, slog` are safe extensions of the usual operators. The code-generator will generate a check for validity and the desired operation, using a newly introduced variable to avoid multiple copies of the expressions.

Note: the usual infix-operators with the usual operator precedence as well as prefix notation (e.g., '+'(X,Y)) can be used.

A.6.1 Boolean Expressions

```
BOOLFUNC ::=
    nonzero ( EXPR )
    | true
    | false
    | EXPR RELOP EXPR

RELOP ::=
    < =< > >= == !=
```

Note: the $\leq$ is `=<` to conform to PROLOG standard.

Note: The operation `nonzero` has been introduced for handling numerical instability. Whereas `EXPR != 0` really checks for being equal to 0, `nonzero(EXPR)` just checks if the absolute value of `EXPR` is larger than some given ϵ.

A.6.2 Numeric expressions and functions

```
NUMFUNC ::=
    sqrt ( EXPR )
    | exp ( EXPR )
    | sin ( EXPR )
    | abs ( EXPR )
    | random
    | random_int(EXPR, EXPR)
```

The function `random` returns a pseudo-random number between 0 and 1; `random_int` returns a pseudo-random integer in the given range.

A.6.3 Summation expression

```
SUM_EXPR ::=
    sum( [ IDX_LIST], STAT ,ATTR)
```

A.6.4 Indexed Expressions

```
IDX_EXPR ::=
    select( IDENT, [ IDX_LIST])
```

A.6.5 Getting the Norm of an iteration

```
NORM_EXPR ::=
    norm( EXPR, [ IDX_LIST], EXPR ,ATTR)
```

The intended meaning of this construct is to get the value of EXPR1 normed to EXPR2. For example,

`norm(v(i),[idx(j,1,N)],v(j),[])` calculates: $v(i)/\sum_{j=1}^{N} v(j)$.

The expression `norm(EXPR,[IDX_LIST],EXPR2)` unfolds into

```
EXPR1 / sum([IDX_LIST],EXPR2)
```

Since usually (or actually the only thing which makes sense) the sum-expression is constant wrt. the EXPR1 (in our case the i), this sum could be moved out of the for-loop.

However, beware of the situation where you have:

```
for ( [idx(i,0,N)],
v(i) = norm(v(i),[idx(j,0,N)],v(j)] )
```

This would NOT correctly normalize that vector (because you modify the v(i) and with that the sum. So care must be taken to take the correct thing.

A.6.6 Maxarg

```
MAXARG_EXPR ::=
    maxarg( [ IDX_LIST], EXPR ,ATTR)
```

determine index where EXPR gets its maximal value.

A.6.7 conditional expressions

```
CONDEXPR ::=
    | cond ( EXPR , EXPR , EXPR )
```

Appendix B. Useful AutoBayes Pragmas

A list of all pragmas formatted in LaTeXcan be generated by `autobayes -tex -help pragmas`. This is a subset.

`cg_comment_style` **(atomic)** select comment style for C/C++ code generator

Default: `-pragma cg_comment_style=cpp`

Possible values :

`kr` use traditional (KR) style comments

`cpp` use C++ style comments //

`cluster_pref` **(atomic)** select algorithm schemas for hidden-variable (clustering) problems

Default: `-pragma cluster_pref=em`

Possible values :

`em` prefer EM algorithm

`no_pref` no preference

`k_means` use k-means algorithm

`codegen_ignore_inconsistent_term` **(boolean)** [DEBUG] ignore inconsistent-term conditional expressions in codegen

Default: `-pragma codegen_ignore_inconsistent_term=false`

`em` **(atomic)** preference for initialization algorithm for EM

Default: `-pragma em=no_pref`

Possible values :

`no_pref` no preference

`center` center initialization

`sharp_class` class-based initialization (sharp)

`fuzzy_class` class-based initialization (fuzzy)

`em_log_likelihood_convergence` **(boolean)** converge on log-likelihood-function

Default: `-pragma em_log_likelihood_convergence=false`

`em_q_output` **(boolean)** Output the Q matrix of the EM algorithm

Default: `-pragma em_q_output=false`

`em_q_update_simple` **(boolean)** force the q-update to just contain the density function

Default: `-pragma em_q_update_simple=false`

`ignore_division_by_zero` **(boolean)** DEBUG: Do not check for X=0 in X**(-1) expressions

Default: `-pragma ignore_division_by_zero=false`

`ignore_zero_base` **(boolean)** DEBUG: Do not check for zero-base in X**Y expressions

Default: `-pragma ignore_zero_base=false`

`infile_cpp_prefix` **(atomic)** Prefix for intermediate input file after cpp(1) processing

Default: `-pragma infile_cpp_prefix=cpp_`

`instrument_convergence_save_ub` **(integer)** default size of instrumentation vector for convergence loops

Default: `-pragma instrument_convergence_save_ub=999`

`lopt` **(boolean)** Turn on/off optimization of the lang code

Default: `-pragma lopt=false`

`optimize_cse` **(boolean)** enable common subexpression elimination

Default: `-pragma optimize_cse=true`

`optimize_expression_inlining` **(boolean)** enable inlining (instead function calls) of goal expressions by schemas

Default: `-pragma optimize_expression_inlining=true`

`optimize_max_unrolling_depth` **(int)** maximal depth of for-loops w/ constant bound to be unrolled

Default: `-pragma optimize_max_unrolling_depth=3`

`optimize_memoization` **(boolean)** enable subexpression-memoization

Default: `-pragma optimize_memoization=true`

`optimize_substitute_constants` **(boolean)** allow values of constants to be substituted into loop bounds

Default: `-pragma optimize_substitute_constants=true`

`rwr_cache_max` **(integer)** size of rewrite cache

Default: `-pragma rwr_cache_max=2048`

`schema_control_arbitrary_init_values` **(boolean)** enable initialization of goal variables w/ arbitrary start/step values

Default: `-pragma schema_control_arbitrary_init_values=false`

`schema_control_init_values` **(atomic)** initialization of goal variables

Default: `-pragma schema_control_init_values=automatic`

Possible values :

`automatic` calculate best values

`arbitrary` use arbitrary values

`user` user provides values (additional input parameters

`schema_control_solve_partial` **(boolean)** enable partial symbolic solutions

Default: `-pragma schema_control_solve_partial=true`

`schema_control_use_generic_optimize` **(boolean)** enable intermediate code generation w/ generic optimize(...)-statements

Default: `-pragma schema_control_use_generic_optimize=false`

`synth_serialize_maxvars` **(integer)** maximal number of solved variables eliminated by serialize

Default: `-pragma synth_serialize_maxvars=0`

Appendix C. Examples

C.1 Simple AutoBayes Problem

C.1.1 Specification

Throughout the text, the following simple AutoBayes specification is used (Listing C.1). Section C.1.2 shows the autogenerated derivation for this problem. The entire LaTeXdocument has been generated except for the red lines.

Notes:

- the original specification uses mu, sigma, etc. The LaTeX output automatically converts most greek names into greek symbols; variable names ending in `_sq` are converted into squares.
- Upper case symbols can be used in specifications, when the flag `prolog_style` is set to false.
- LaTeX output is produced using the `-tex synt` command-line option.
- The current version type-sets the entire program (code and comments); for the derivation below, only the comments were extracted (manually).

```
1  model normal_simple as 'Normal model without priors'.
2
3  double mu.
4  double sigma_sq as 'sigma squared'.
5    where 0 < sigma_sq.
6
7  const nat n as '# data points'.
8    where 0 < n.
9
10 data double x(0..n-1) as 'known data points'.
11 x(_) ~ gauss(mu, sqrt(sigma_sq)).
12
13 max pr(x | {mu, sigma_sq}) for {mu, sigma_sq}.
```

Listing C.1: Simple AutoBayes specification

C.1.2 Autogenerated Derivation

begin autogenerated `max pr(x|mu,s) for mu,s`

The conditional probability $\mathbf{P}(x \mid \mu, \sigma^2)$ is under the dependencies given in the model equivalent to

$$\prod_{i=0}^{-1+n} \mathbf{P}(x_i|\mu, \sigma^2)$$

schema: prob-2-formula

The probability occurring here is atomic and can thus be replaced by the respective probability density function given in the model. This yields the log-likelihood function PDF(gauss) = 1/..* exp(...)

$$\log \prod_{i=0}^{-1+n} \exp\left(\frac{-\frac{1}{2}\,(x_i-\mu)^2}{(\sigma^2)^{\frac{1}{2}2}}\right) \frac{1}{\sqrt{2\,\pi}\,(\sigma^2)^{\frac{1}{2}}}$$

which can be simplified to

$$-\frac{1}{2}\,n\,\log 2 + -\frac{1}{2}\,n\,\log \pi + -\frac{1}{2}\,n\,\log \sigma^2 + -\frac{1}{2}\,(\sigma^2)^{-1} \sum_{i=0}^{-1+n} (-1\,\mu + x_i)^2$$

This function is then optimized w.r.t. the goal variables μ and σ^2. optimization solves the maximation task

The summands

$$-\frac{1}{2}\,n\,\log 2$$
$$-\frac{1}{2}\,n\,\log \pi$$

are constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The factor

$$\frac{1}{2}$$

is non-negative and constant with respect to the goal variables μ and σ^2 and can thus be ignored for maximization.

The function

$$-1\, n\, \log \sigma^2 + -1\, (\sigma^2)^{-1} \sum_{i=0}^{-1+n} (-1\, \mu + x_i)^2$$

is then symbolically maximized w.r.t. the goal variables μ and σ^2. The partial differentials text-book like: set first derivative = 0 and solve

$$\begin{aligned} \frac{\partial f}{\partial \mu} &= -2\, \mu\, n\, (\sigma^2)^{-1} + 2\, (\sigma^2)^{-1} \sum_{i=0}^{-1+n} x_i \\ \frac{\partial f}{\partial \sigma^2} &= -1\, n\, (\sigma^2)^{-1} + (\sigma^2)^{-2} \sum_{i=0}^{-1+n} (-1\, \mu + x_i)^2 \end{aligned}$$

are set to zero; these equations yield the solutions

solver can symbolically solve

$$\begin{aligned} \mu &= n^{-1} \sum_{i=0}^{-1+n} x_i \\ \sigma^2 &= n^{-1} \sum_{i=0}^{-1+n} (-1\, \mu + x_i)^2 \end{aligned}$$

end autogenerated document

Appendix D. Exercises

D.1 Running AutoBayes

D.1.1 Exercise 1

Run the norm.ab example and inspect generated code and derivation. If possible, generate the latex version of the derivation.

```
1 model normal_simple as 'Normal model without priors'.
2
3 double mu.
4 double sigma_sq as 'sigma squared'.
5   where 0 < sigma_sq.
6
7 const nat n as '# data points'.
8   where 0 < n.
9
10 data double x(0..n-1) as 'known data points'.
11 x(_) ~ gauss(mu, sqrt(sigma_sq)).
12
13 max pr(x | {mu, sigma_sq}) for {mu, sigma_sq}.
```

Listing D.1: norm.ab

D.1.2 Exercise 2

Generate multiple versions for this problem. Note: use the appropriate flags to allow AutoBayes to generate numerical optimization algorithms:
(`schema_control_arbitrary_init_values`)

D.1.3 Exercise 3

Modify the "norm" example to use a different probability density function. Note that some of them do have a different number of parameters. Inspect the generated code and derivation. Can the problem be solved symbolically for all PDFs?

Hint: use vonmises1, poisson, weibull, cauchy

D.1.4 Exercise 4

Generate multiple versions for the mixture-of-gaussians example. What are the major differences between the different synthesized programs.

Note: the specification is `mog.ab` in the models_manual directory.

Generate a sampling data generator (autobayes -sample) for this specification.

In AutoBayes generate 1000 data points that go into 3 different classes. Then run the different programs and see how good they estimate the parameters.

Note: the generated functions require column-vectors, so, e.g., give the means as `[1,2,3]'`

```
octave-3.4.0:1> sample_mog
usage: [vector c, vector x] = sample_mog(vector mu, int n_points, vector
    phi, vector sigma)

octave-3.4.0:2> [c,x] = sample_mog
    ([1,2,4]',1000,[0.3,0.1,0.6]',[0.1,0.1,0.2]');
```

Listing D.2: calling the synthesized code in Octave

D.1.5 Exercise 5

Run a change-point detection model (e.g., `climb_transition.ab` and look at generated code and derivation. How does AutoBayes find the maximum?

D.1.6 Exercise 6

Add the Pareto distribution to the built-in transitions. Get the formulas from wikipedia.

Try the following simple model:

```
model pareto as 'Normal model without priors'.

double alpha.
        where 3 < alpha.
const double xm.
  where 0 < xm.

const nat n as '# data points'.
  where 0 < n.

data double x(0..n-1) as 'known data points'.
        where 0 < x(_).
```

```
13          where xm < x(_).
14
15 x(_) ~ pareto(xm, alpha).
16
17 max pr(x | {alpha}) for {alpha}.
```

Listing D.3: Specification for Pareto distribution

```
1 octave-3.4.0:2> xm=5;
2 octave-3.4.0:3> alpha=15;
3 octave-3.4.0:4> x=xm*(1./(1-rand(10000,1)).^(1/alpha));
4 octave-3.4.0:5> alpha_est = pareto(x,5)
5 alpha_est =  15.081
```

Listing D.4: Generate Pareto-distributed random numbers

Appendix E. Research Challenges and Programming Tasks

E.1 PDFs

E.1.1 Integrate χ^2 PDF into AutoBayes

The χ^2 PDF is important to handle square errors of Gaussian distributed data. E.g., for $X, Y \sim N(\mu, \sigma^2)$ we get $X^2 + Y^2 \sim \chi^2(1)$.

E.1.2 Integrate folded Gaussian PDF into AutoBayes

Folded Gaussian PDF is important to handle problems with abs functions. For $X \sim N(0, 1)$, we get $|X| \sim N_f(\theta)$.

E.1.3 Integrate Tabular PDF into AutoBayes

Handling of non-functional PDFs, e.g., for ground-cover clustering. The PDF is given as a vector over the data X, e.g., as $X \sim tab(p)$ where $constdoublep(0..n-1)$. and $where0 = sum(I := 0..n-1, p(I)) - 1$

Normalization is important

E.2 Gaussian with full covariance

Currently, AutoBayes can only handle Gaussian distribution with a diagonal covariance matrix, i.e., $\Sigma_{i,j} = 0$ for $i \neq j$.

This could be implemented as a separate PDF, or the dimensionality could be inferred from the declaration of the sigmas.

Requires 3-dim arrays for multivariate clustering.

E.3 Preprocessing

E.3.1 Normalization of Data

Develop a schema for the normalization of data toward 0..1 or $N(0,1)$. For Gaussian PDF, $aX + b \sim N(a\mu + b, a^2\sigma^2)$

E.3.2 PCA for multivariate data

This preprocessing cuts down the number of dimensions given by a given goal (threshold on the eigen values or reduction of dimensions). The rnd-projection schema could be used for this.

Note that after clustering, the resulting parameters must be mapped back to the original space.

E.4 Clustering

E.4.1 KD-tree Schema

must be dug up and integrated (Alex Grey(?))

E.4.2 EM schema with empty classes

The current EM algorithm fails if one or more classes become empty. The EM schema must be extended to enable handling this case. Since we cannot dynamically resize the data structures, an index vector (e.g., valid_classes) must be carried along. Refactoring of the EM schema might be a good idea

E.4.3 Clustering with unknown number of classes

With a very simple approach, a schema is developed, which executes a for loop over the number of classes and returns the parameters for the run with the maximum likelihood.

The spec gives the range of class numbers.

Extensions: run the algorithm multiple times (with different random initializations) for each n_classes.

Should be combined with different quality-of-clustering metrics

E.4.4 Quality-of-clustering metrics

Currently, AutoBayes stops clustering, when a given tolerance is reached. Then it returns the log-likelihood as the only quality metric.

The literature describes a large number of different quality metrics for clustering. Develop schemas for calculating one or more of these metrics after each run of a clustering algorithm

E.4.5 Regression Models

E.5 Specification Language

E.5.1 Improved Error Handling

E.6 Code Generation

E.6.1 R Backend

R is a popular language for statistics purposes. Thus an interface of AutoBayes to R is important to increase the usability of AutoBayes.

E.6.2 Arrays in Matlab

Currently all matrices and arrays are linearized and the access is done using a macro. I.e., $x_{i,j}$ is implemented as `*(x+i*N +j)`.

The access with using a vectorized linearization is to be implemented.

E.6.3 Java Backend

Develop a backend for stand-alone Java. This requires definition of a suitable data structure for arrays (and their allocation), handling of multiple arguments and return values).

E.6.4 C stand-alone

The C stand-alone code generator must be debugged and improved

E.6.5 Code Generator Extensions for functions/procedures

E.7 Numerical Optimization

E.7.1 GPL library

E.7.2 Multivariate Optimization

Add schema-library for multivariate optimization

E.7.3 Optimizations under Constraints

trust region algorithms

E.8 Symbolic

E.8.1 Handling of Constraints

E.9 Internal Clean-ups

E.9.1 Code generation

cg_compoundexpr.pl

E.10 Major Debugging

E.10.1 Fix all Kalman-oriented examples

E.11 Schema Control Language

E.12 Schema Surface Language

E.12.1 Domain-specific surface language for schemas

E.12.2 Visualization of Schema-hierarchy

E.12.3 Schema debugging and Development Environment

E.13 AutoBayes QA

E.14 AutoBayes/AutoFilter

Implementation of Particle Filters for Health Management