MOSES - Quick Start

Predrag Janicic, Revised by Linas Vepstas

3 November 2008, revised 14 December 2011

Abstract

Meta-optimizing semantic evolutionary search (MOSES) is a new approach to program evolution, based on representation-building and probabilistic modeling. MOSES has been successfully applied to solve hard problems in domains such as computational biology, sentiment evaluation, and agent control. Results tend to be more accurate, and require less objective function evaluations, than other program evolution systems, such as genetic programming or evolutionary programming . Best of all, the result of running MOSES is not a large nested structure or numerical vector, but a compact and comprehensible program written in a simple Lisp-like mini-language.

Contents

1   Introduction

Meta-optimizing semantic evolutionary search (MOSES) is a new approach to program evolution, based on representation-building and probabilistic modeling. MOSES has been successfully applied to solve hard problems in domains such as computational biology, sentiment evaluation, and agent control. Results tend to be more accurate, and require less objective function evaluations, in comparison to other program evolution systems. Best of all, the result of running MOSES is not a large nested structure or numerical vector, but a compact and comprehensible program written in a simple Lisp-like mini-language. For more information see

http://metacog.org/doc.html.

The list of publications on MOSES is given in the References section. Moshe Look's PhD thesis [6] should be the first material to be read by someone interested in using or modifying MOSES.

MOSES is implemented in C++ and it heavily uses templates. So, one interested in modifying MOSES must be familiar with C++ and, at least to some extent, to C++ templates.

2   Copyright Notice

MOSES is Copyright 2005-2008, Moshe Looks and Novamente LLC.

It is licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

3   Installation

To compile MOSES, you need

- a recent gcc (4.x or later);

- the boost libraries (http://www.boost.org/);

- the CMake package (http://www.cmake.org/HTML/Index.html);

For compiling MOSES, create a directory `build` (from the root folder of the MOSES distribution), go under it and run "`cmake ..`". This will create the needed build files. Then, make the project using "`make`" (again from the directory `build`). Generated executables will be in the folder `build/moses/learning/moses/main`.

4 Overview

MOSES performs supervised learning, and thus requires either a scoring function or training data to be specified as input. As output, it generates a Combo program that, when executed, approximates the scoring function. MOSES uses general concepts from evolutionary search, in that it maintains a population of programs, and then explores the neighborhood of modified, "mutated" programs, evaluating their fitness. After some number of iterations, the fittest program found is output.

More precisely, MOSES maintains a population of demes. Each deme is a program with many adjustable, tuneable parameters. These adjustable parameters are colloquially referred to as knobs. Thus, finding the fittest program requires not only selecting a deme, but also determining the best setting for the knobs.

The MOSES algorithm proceeds by selecting a deme and performing random mutations on it, by inserting new knobs in various random places. The best-possible knob settings for the mutated deme are found by using using existing, well-known optimization algorithms such as hill-climbing, simulated annealing or estimation of distribution algorithms (EDA) such as Bayesian optimization (BOA/hBOA). The fitness of the resulting program(s) can be compared to the fittest exemplar of other demes. If the new program is found to be more fit, it is used to start a new deme, (while discarding the old, dominated deme) and the process then repeats.

All program evolution algorithms tend to produce bloated, convoluted, redundant programs ("spaghetti code"). To avoid this, MOSES performs reduction at each stage, to bring the program into normal form. The specific normalization used is based on Holman's "elegant normal form", which mixes alternate layers of linear and non-linear operators. The resulting form is far more compact than, say, for example, boolean disjunctive normal form. Normalization eliminates redundant terms, and tends to make the resulting code both more human-readable, and faster to execute.

The above two techniques, optimization and normalization, allow MOSES to out-perform standard genetic programming systems. The EDA algorithms, by finding the dependencies in a Bayesian network, in fact are able to find how different parts of a program are related. This quickly rules out pointless mutations that change one part of a program without making corresponding changes in other, related parts of the program. The other important ingredient, reduction to normal form, allows programs to become smaller, more compact, faster to execute, and more human readable. Besides avoiding spaghetti code, normalization removes redundancies in programs, thus allowing smaller populations of less complex programs, speeding convergence.

The programs that MOSES generates are "generic", in the sense that MOSES works with structured trees, represented in Combo. Such trees can represent propositional formula, procedural or functional programs, etc. The core MOSES solver is written in C++, and takes the form of a library. There are many example programs illustrating how to use this library.

5 Terminology

MOSES uses a vocabulary specific to itself. Some of the most important terms are defined below.

Program. A *program*, in MOSES, is represented as a tree of operators, variables and values. Nodes in the tree may be constants (bits, integers, real numbers, *etc.*), boolean operators (*and, or, etc.*), arithmetical operators (+, -, *, *etc.*), functions (*sin, cos, etc.*) or logical expressions (*if...then...else, etc.*), and so on. Arguments to an n-ary function are denoted with hash marks, so that $\#1, \#2, \ldots, \#n$ would be the arguments. Thus, for example, $(0 < (0.5 * \#1)) \vee \#2$ is a program that takes argument $\#1$ (a float pt. number), multiplies it by 0.5, and checks to see if it is greater than zero. The result of this compare is or-ed ($\vee$-ed) with argument $\#2$ (a boolean). Although programs may be explicit, as in this example, a program can also be understood to be a *representation*, together with a particular set of *knob settings*, as explained below.

Exemplar. An *examplar* is a specific program; typically, the fittest one found.

Representation. A representation is a parameterized tree structure, representing a particular region of program space, centered around a single program (the *exemplar*). A representation is derived from the examplar by inserting additional nodes in various (random) locations. The inserted nodes, however, are not specific values or functions or operators, but are rather place-holders for values/functions to be determined later. Each place-holder may be thought of as a *parameter*, and is colloquially referred to as a *knob*. Thus, a program is a representation, together with a particular setting of the knobs. During the optimization step in MOSES, the space of all possible parameter or knob settings will be explored, to locate the best possible settings, *i.e.* to find the fittest program.

Knobs. A *knob* is a single dimension of variation relative to a representation tree. It may be discrete or continuous. For example, given the program tree fragment $(0 < (0.5 * \#1)) \vee \#2$, a continuous knob might be used to vary the numerical constant 0.5 to other values. So, setting this knob to 0.7 would transform this tree fragment to $(0 < (0.7 * \#1)) \vee \#2$. Discrete knobs have a '*multiplicity*': the number of different possible settings they may have. Continuous knobs have an effectively infinite multiplicity.

A discrete knob with a multiplicity of 4 might be used to transform the boolean input $\#2$, with 0 meaning 'always true', 1 meaning 'invert', 2 meaning 'don't invert' and 3 meaning 'always false'. So, setting this knob to 1 would transform the above example tree to $(0 < (0.5 * \#1)) \vee (\neg \#2)$. A discrete knob of multiplicity 6 might be used to replace the less-than comparison with $\leq, >, \geq, =$or $\neq$(thus making six possible comparison operators, for a multiplicity of 6 for this knob). Another discrete knob of

multiplicity 3 might replace the 'or' symbol $\vee$ with the 'and' symbol $\wedge$ or the 'exclusive-or' symbol $\otimes$. Knobs do not have to be defined as running over all possible values; it is usually convenient to keep multiplicity fairly low. This will usually help avoid excess redundancy in program space, although resulting programs may be more verbose.

Representation-building. The step in the MOSES algorithm where an examplar is chosen, and a representation is constructed from it. This representation is essentially the cornerstone of a deme.

Deme. A *deme* is a population of programs derived from one single representation. Thus, a deme can be thought of as a population of knob settings. During the optimization phase, an optimizer algorithm, such as hill-climbing, simulated annealing, or the Bayesian optimization algorithm is used to work with the population, locating the best possible knob settings for the given representation.

Metapopulation. MOSES maintains a collection of demes, playing each off the others. This set of demes is referred to as the *metapopulation*. Pairs of demes are in competition; fitter demes are used to replace less fit demes.

Scoring function. During the optimization phase, candidate programs being explored are scored by a *scoring function*. The function is specific to the given problem; it returns a value indicating how closely the candidate program matched the desired output. For supervised training problems, the scoring function just returns how closely the candidate program matched the training set. For demonstration problems, the scoring function is typically some well-studied toy problem, such as parity, santa-fe-trail, *etc.* Usually, the perfect score is 0, while worse scores are negative. Fitter programs have higher scores.

Domination. One deme is considered to *dominate* another if it is better in every way. During learning, demes typically do not dominate one-another: some are better at some parts of the scoring function, while others are better at other aspects. Thus, both are kept around and further evolved. Demes that are completely dominated are (usually) discarded.

Normalization. The *normalization* step of the MOSES algorithm takes a program, and simplifies it, using *re-writing rules*. The resulting program is said to be in *normal form*. Thus, for example, $\#3 \vee F$ can be reduced to just $\#3$ since or-ing with false changes nothing. Similarly, $0 < 0.5 * \#6$ can be normalized to $0 < \#6$ since multiplying by one-half never changes the sign of a number. Likewise, the expression "if $(x = x)$ then y" can be reduced to y, since a value is always equal to itself, and so the if-branch is always taken. Normalization can sometimes eliminate large parts of a program, if they are vacuous or tautological. There are many different types of normalization that are possible; MOSES always normalizes to the so-called '*elegant normal form*'.

6   MOSES Algorithm

The basic MOSES algorithm can be described as follows:

1. Choose one or more exemplars, expressed as program trees. Initially, this may be the empty program.

2. Representation-building step. For each exemplar, construct a representation; that is, a set of knobs based on the chosen exemplar. Generate a random sampling of knob values, i.e. of programs based on the representation. This is a deme; add this deme to the metapopulation, possibly displacing a less promising deme.

3. Select a deme from the metapopulation and iteratively update its sample, as follows:

   (a) Select some promising programs from the deme's existing sample to use for modeling, according to the scoring function. Ties in the scoring function are broken by preferring smaller programs.

   (b) Considering the promising programs as collections of knob settings, generate new collections of knob settings by applying some (competent) optimization algorithm.

   (c) Convert the new collections of knob settings into their corresponding programs, evaluate their scores, and integrate them into the deme's sample, replacing less promising programs.

4. For each new program that meets the criteria for creating a new deme, go to step 2.

Note that representation-building, the optimization algorithm, and normalization are vital components of the system, and they crucially influence its performance. Representation building is specific for each domain (e.g., learning propositional formulae), while the optimization algorithm is general. MOSES currently supports representation building for several problem domains, including propositional formulae, actions[1], and other things (XXX like what??). MOSES also supports several different optimization algorithms, including hill-climbing, simulated annealing and Bayesian optimization. Work on support-vector machine (SVM) optimization is underway. Only one form of program reduction, to elegant normal form, is supported. Other types of reduction, e.g. SAT-based or satisfiability-modulo-theory (SMT) may be possible but remains unexplored.

   XXX todo: Above algorithm is not quite correct. In particular, the normalization step is omitted.

---

[1]By "actions" we mean mini programming languages describing actions of a agents such as artificial and [6]. Available actions typically cover atomic instructions like "step forward", "rotate left", "rotate right", "step forward", branching instruction such as „if-then-else", and loop instruction such as „while".

7 Source Files and Folders

In the MOSES distribution, there are the following folders (under `moses/learning/moses`) with the source files:

eda This folder contains support for estimation of distribution algorithms, and the lower level support for optimization algorithms.

example-progs This folder contains examples demonstrating different features of the MOSES system: reducing expressions, and, of course, applications of MOSES itself. It includes examples for the "ant problem/santa-fe trail", for "parity formulae", *etc.*

main Contains the main moses executable.

moses This folder contains the core support for MOSES – including base type definitions, and distributed computation support.

optimization Contains the main optimization code.

representation Contains code for representation-building.

8 Types, Structures, and Classes

This section briefly reviews some of the key datatypes and classes found in the code. MOSES makes heavy use of C++ templating. This is done so as to avoid the need for defining base classes, and so avoid the need for pervasive use of derived classes and virtual methods.

8.1 Structured expression trees

For representing structured expression trees (programs, propositional formulae, *etc.*) MOSES relies on the library ComboReduct. In this library, structured expressions are represented by trees of the type `combo tree` as follows:

```
typedef Util::tree<vertex> combo_tree;
```

The file `comboreduct/combo/vertex.h` defines `vertex` as shown blow. It is done this way so that it can capture different sorts of nodes, for different, but still fixed, problem domains.

```
typedef boost::variant<builtin,
      wildcard,
      argument,
      contint,
      action,
      builtinaction,
      perception,
      definiteobject,
```

```
      indefiniteobject ,
      message ,
      procedurecall ,
      anntype ,
      actionsymbol > vertex ;
```

For more information, review the docs provided in the distribution of the Com-
boReduct library.

8.2   Representation building

Conceptually, representation building proceeds by selecting a single program
tree or exemplar, and then building a neighborhood of nearby programs. The
result is a deme: a population of similar programs, centered upon the exemplar.
After representation building, the optimization step is performed, to find the
fittest programs in the deme.

Representation building proceeds in several steps: the exemplar is festooned
with a set of 'knobs': these are adjustable parameters added to the program
tree, to alter its operation. An exemplar festtoned with knobs is called a
'representation'. A deme is then constructed: the deme consists of a pop-
ulation of different possible knob settings. Each of these steps is reviewed in
greater detail, below.

The representation itself is specified by the class representation, given in
representation/representation.h. The structure itself maintains a copy of
the exemplar, as well as the collection of knobs used to create the representation.
Knobs are added to the exemplar by the class constructor.

This class, however, does not maintain the set of possible knob settings;
these are the deme, and are held elsewhere.

8.3   Knobs

Knobs represent tunable parameters in a representation. In the MOSES imple-
mentation, every knob is defined with respect to a particular program tree; that
is, ever knob has a specific, explicit location in a specific, explicit combo_tree.
Knobs are defined in representation/knobs.h and inherit from the base class
knob_base:

```
struct knob_base {
   protected :
      combo_tree& _tr ;
      combo_tree :: iterator _loc ; // location of knob in tree
};
```

Knobs may be continuous (contin_knob) or discrete. Discrete knobs have
a fixed number of settings, determined at compile time:

```
template < int Multiplicity >
struct discrete_knob : public knob_base {...};
```

although some of the discrete knob settings may be disallowed at runtime, effectively decreasing the total multiplicity.

In general, knobs may be "present" or "absent". XXX The semantics of this is unclear. What does this mean? Knobs also have "default" settings, but the meaning of this setting is also unclear, as the code does wonky things with this... XXX TODO fix this.

The `logical_subtree_knob` is suitable for propositional formulae; it has three possible settings: *present*, *absent*, or *negated*. In `simple_action_subtree_knob` knobs, a subexpression can be just *present* or *absent*. In action knobs, a node can have different settings, corresponding to atomic or compound actions, sampled as "perms" in the method `build_knobs::sample_action_perms`. XXX more explanation please ...

Knobs are added to an exemplar using `class build_knobs` defined in `representation/build_knobs.h`.

8.4   Packed Knobs

In order for knob settings to be efficiently managed by the optimization step, they are packed into a bit-string; the bit-string is of type `instance`, declared in `eda/eda.h` as a vector of ints:

```
typedef unsigned int packed_t;
typedef vector<packed_t> instance;
```

The class `field_set` describes the manner in which knobs are packed into the bit string. Defined in `representation/field_set.h`, it contains a list of fields with describe the width of a field, and it's offset within the bit-string:

```
struct field {
   width_t width;
   size_t major_offset, minor_offset;
};
```

Fields come in four basic types: continuous, discrete, "ontological", and single-bit booleans. Discrete fields represent variables that can take on multiple distinct values. These typically take several bits to represent; single-bit boolean variables are treated distinctly. The "ontological" type is simply a discrete type that takes values over a set of character strings. Continuous variables can take on values in the real-number line. However, they are not represented by floats or doubles, but rather, are represented with a certain binary tree of intervals of the real-number line. This encoding is used in order to avoid various difficulties that optimization algorithms encounter with floats and doubles. The rationale and design is further discussed here: http://code.google.com/p/moses/wiki/ModelingAtomSpaces.

The `field_set` provides a set of iterators for walking over these four types; the iterators can be used to extract and change specific values in the bit-string, in the usual fashion. That is, dereferencing an iterator gives the value. So for example, the `disc_iterator` can be used to iterate over discrete fields in an instance:

```
const_disc_iterator begin_disc(const instance& inst) const;
const_disc_iterator end_disc(const instance& inst) const;

instance_t my_inst = ...;
const_disc_iterator it = begin_disc(my_inst);
for ( ; it<end_disc(my_inst); it++) {
    cout " This is the value: " << *it << endl;
}
```

Typically, many different instances can be described by the same `field_set`. Thus, when an `instance_set` is defined (see below), one copy of the `field_set` is kept, to describe all of the instances in an `instance_set`.

These unpacked fields are associated with actual knobs in a specific `combo_tree` by means of the `knob_mapper`. So, for example

```
typedef std::multimap<field_set::disc_spec, disc_knob> disc_map;
```

is a map between `disc_knob`'s (discrete knobs, which are aware of the `combo_tree` in which they are located), and `disc_spec`'s (knob specifications, which are used by `field_set` to describe fields in the packed bit-string). This map can be used to find a knob value in a packed bit-string, or, conversely, given a field, to find the corresponding knob in a `combo_tree`.

8.5   Representation, revisited

XXX some details below incorrect.

The constructor of this structure, builds knobs with respect to the given exemplar (by the method `build_knobs`).

This structure stores the exemplar like a tree (more precisely `combo_tree`). This structure has a method for using a given instance to transform the exemplar (`transform`) providing a new expression tree.

The structure also has methods for clearing the current version of the exemplar (setting all knobs to default values — zeros) — `clear_exemplar`, for getting the exemplar — `get_clean_exemplar`, and for getting the reduced, simplified version of the exemplar `get_clean_exemplar`.

8.6   Scoring

The fitness of program trees are ranked with scores. There are several types of scores, these are all defined in `moses/types.h`.   The most basic is

```
typedef float score_t;
```

Different programs may be code at different things, and so judging their fitness in multiple ways requires a vector:

```
typedef std::vector<score_t> behavioral_score;
```

Programs are also scored according to their complexity, so in `moses/complexity.h` we find:

```
typedef int complexity_t;
```

Composite scores pair up the complexity and the fitness score:

```
typedef std::pair<score_t, complexity_t> composite_score;
```

while behavioral composite scores combine the complexity measure with the vector:

```
typedef tagged_item<behavioral_score, composite_score>
   composite_behavioral_score;
```

8.7 Demes and Optimization

At the conceptual level, the optimization algorithms operate on a population in a deme, by twiddling knobs until the optimizer finds a program that performs well. That is, a deme is a collection of knob settings: for evolutionary optimization algorithms, it may be thought of as a population: different instances of knob settings compete with one-another until a population of the best-possible knob settings is found.

At the practical level, optimization operates on a population of bit-strings, and not on abstract 'knob settings'. Thus, a necessary step of the MOSES algorithm is to convert such abstract knob settings into packed bit strings, and *vice versa*. This is done during the representation step of the algorithm: knobs were inserted into an exemplar, the knobs were mapped to knob specifications, and the knob specifications were in-turn mapped to bit-strings. The optimization algorithm then tries to find the fittest bit-strings or 'instances'.

So, in eda/scoring.h, we find scored instances. The higher the score, the fitter the bit-string:

```
template<typename ScoreT>
struct scored_instance :
   public tagged_item<instance, ScoreT> {...};
```

In representation/instance_set.h we find a collection of scored instances:

```
template<typename ScoreT>
struct instance_set :
   public vector<scored_instance<ScoreT> > {
   ...
   protected:
      const field_set &_fields;
};
```

Note that, in the above, there is a field_set member used to describe how the packed bits in the instance are to be unpacked.

Optimization works on these, so in eda/optimize.h we find a function template:

```
template <...>
int optimize(instance_set<ScoreT>&, ...);
```

The output of the optimization step is a deme: conceptually, a population of the fittest programs there were centered on the exemplar.

8.8   Metapopulation

The metapopulation is a set of scored combo trees. More precisely, they are scored with composite behavioural scores (or b-scores). A 'bscored combo tree' is then just a pair that associates a b-score with a tree, defined in the file `moses/types.h`.

```
typedef tagged_item < combo :: combo_tree ,
        composite_behavioral_score > bscored_combo_tree ;
```

The metapopulation is then a set of scored combo trees, defined in `moses/metapopulation.h`. More precisely, it is a template, inheriting from the set:

```
typedef std :: set < bscored_combo_tree > bscored_combo_tree_set ;


template < typename Scoring ,
          typename BScoring ,
          typename Optimization >
struct metapopulation :
    public bscored_combo_tree_set {...};
```

The template plays only a small role in this class; it's only purpose is to allow generic scoring and optimization algorithms to be used with the metapopulation.

The metapopulation will store expressions (as scored trees) that were encountered during the learning process (not all of them; the weak ones, which are dominated by existing ones, are usually skipped as non-promising).

As an example, one can iterate through the metapopulation and print all its elements with their scores and complexities in the following way:

```
for (const iterator it=begin(); it!=end(); ++it)
   cout << gettree(*it) << " "
      << getscore(*it) << " "
      << getcomplexity(*it) << endl;
```

The metapopulation is updated in iterations. In each iteration, one of its elements is selected as an exemplar. The exemplar is then used for building a new deme (that will further extend the metapopulation).

9   Key Methods

This section briefly discusses key MOSES's methods.

9.1   Representation Building

The creation of representations from exemplars, termed '*representation building*', is one of the key ideas of the MOSES algorithm. Representation building

13

starts with an exemplar and adds to it new nodes and corresponding knobs (see [6]). Knobs can have different settings. If all knobs are set to 0, then the original exemplar is obtained. The code provides support for several different problem domains, including propositional formulae and actions. These are given in the file `representation/build_knobs.h`.

The key method is `void build_knobs::build_action(pre_it it)` — it substantially determines the representation building for one domain and substantially influence the learning process.

9.2 Optimization

The role of optimization is to score elements of the deme, and to process them and to generate new promising instances. The optimization process works over sequences of numbers (i.e., over "instances"). It is invoked in the main procedure, in each iteration of expanding (the method `metapopulation::expand`, implemented in the file `moses/moses.h`) the metapopulation. The result is a set of instances that are transformed to trees and then added to the metapopulation (if they are not dominated by existing elements).

Currently, there are two optimization methods implemented (implemented in the file `moses/optimization.h`):

univariate optimization based on Bayesian optimization (see [6]);

iterative hillclimbing based on a simple greedy iterative process, looking for a better than exemplar instance at distance $1, 2, 3, \ldots$;

In addition, there is support for "sliced iterative hillclimbing", similar as the one as above, but using a time slicing, so it can be used in some wider context, without leaving other subtasks idle for too long.

Optimization algorithms also have to take care of number of evaluations (number of calls to scoring functions) used. Basically, this number controls the resources given to the algorithm.

9.3 Scoring

While the representation building is specific for each problem domain, the scoring is specific for each specific problem. For instance, in learning propositional formulae, the same representation building algorithm is used, but different scoring functions will be used for each specific task (for instance, for learning disjunction or conjunction over the given set of propositional variables).

There is a

score function returning `int`, generally with 0 as a perfect score, and negative numbers as worse scores;

behavioral score function returning a vector of specific values, that is used for comparing expressions on different dimensions and for discarding elements dominated by other elements.

Technically, these functions are provided as operators within structures.

The scoring functions are used for instantiating higher-level functions used uniformly for different problem domains and different problems.

Some low-level, problem specific, scoring functions are defined in the file (under `src/MosesEda`) `moses/scoring_functions.h`, while higher-level support is defined in the file `moses/scoring.h/cc`.

10 MOSES: Putting It All Together

With all components briefly described above, this section discusses how are they combined in a system MOSES.

The main moses method is trivial: it only expand the metapopulation in iterations until the given number of evaluations or a perfect solution is reached. This method is implemented in `moses/moses.h`, in several variations (some with additional arguments corresponding to available actions and perceptions, just for the action problem domain).

Typical usage of MOSES starts by providing scoring functions. For instance, for learning disjunction propositional formula one can use the following declaration (defined in `moses/scoring_functions.h`):

```
disjunction scorer;
```

and for solving the ant problem, one can use the following declaration (defined in `moses/scoring_functions.h`):

```
antscore scorer;
```

Also, the type of expression to be learnt has to be provided [2]. For instance, for the disjunctive formula, one should use:

```
typetree tt(id::lambdatype); tt.appendchildren(tt.begin(),id::booleantype,arity+1);
```

where `arity` carries the information of the number of propositional variables to be considered. For the ant problem, one would write:

```
typetree tt(id::lambdatype); tt.appendchildren(tt.begin(),id::actionresulttype,1);
```

Then the metapopulation has to be declared. It is instantiated via templates, saying which scoring function, which behavioral scoring function, and which optimization algorithm to use. As, arguments one has to provide the random generator, the initial exemplar, the type tree, simplification procedure, then the scorers and the optimization algorithm. This is an example for learning the disjunctive formula:

---

[2]for a detail explanation of the type system used in ComboReduct see the doc provided with the distribution of ComboReduct

```
 metapopulation<logicalscore,logicalbscore,univariateoptimization>
metapop(rng, vtree(id::logicaland),tt,logicalreduction(), logicalscore(scorer,arity,rng),
logicalbscore(scorer,arity,rng), univariateoptimization(rng));
```

and this is an example for the ant problem:

```
 metapopulation<antscore,antbscore,univariateoptimization>
metapop(rng,vtree(id::sequentialand),tt,actionreduction(), scorer,
bscorer, univariateoptimization(rng));
```

There is also a version of MOSES that uses the sliced interactive hillclimbing — `sliced_moses`. It supports the action domain, but can be simply modify to support other domains as well.

These, and several more examples, can be found in the folder `main`. The example programs provided often ask for arguments like the seed for the pseudo-random number generation or for the number of evaluations.

11 Final Remarks

While MOSES is not that big a system, it cannot be documented in detail in just a few pages. However, the descriptions given above should be helpful when one first encounters MOSES and tries to use it and modify it.

Currently, MOSES together with ComboReduct consists of 17 KLOC of .cc files and 24 KLOC of header files, as counted by the wc command. This includes all comments, copyright notices, example programs and utilities. Of this, combo consists of about 18 KLOC while MOSES consists of 22 KLOC.

References

[1] Moshe Looks, "Scalable Estimation-of-Distribution Program Evolution", Genetic and Evolutionary Computation Conference (GECCO), 2007.

[2] Moshe Looks, "On the Behavioral Diversity of Random Programs", Genetic and Evolutionary Computation Conference (GECCO), 2007.

[3] Moshe Looks, "Meta-Optimizing Semantic Evolutionary Search", Genetic and Evolutionary Computation Conference (GECCO), 2007.

[4] Moshe Looks, Ben Goertzel, Lucio de Souza Coelho, Mauricio Mudado, and Cassio Pennachin,"Clustering Gene Expression Data via Mining Ensembles of Classification Rules Evolved Using MOSES", Genetic and Evolutionary Computation Conference (GECCO), 2007.

[5] Moshe Looks, Ben Goertzel, Lucio de Souza Coelho, Mauricio Mudado, and Cassio Pennachin, "Understanding Microarray Data through Applying Competent Program Evolution", Genetic and Evolutionary Computation Conference (GECCO), 2007.

[6] Moshe Looks, "Competent Program Evolution" Doctoral Dissertation, Washington University in St. Louis, 2006.

[7] Moshe Looks, "Program Evolution for General Intelligence", Artificial General Intelligence Research Institute Workshop (AGIRI), 2006.