MaLiJAn Manual

Florian Furbach Michael Holzhauser
Raphael Reitzig Vasil Tenev Sebastian Wild

September 28, 2012

Contents

1 Overview **3**
1.1 Steps of Analysis — A Glossary 3

2 Primer — Theorectical Foundations **6**
2.1 Maximum Likelihood Analysis 6
2.2 The Art of Extrapolation: How to get $p_{i \to j}(n)$? 7

3 Prepare Your Algorithm **8**
3.1 In-code cost measures 8
3.2 Compiling 8
3.3 Unsupported Language Constructs 9

4 Analyze Your Algorithm **10**
4.1 Creating a new project 10
4.2 Loading a project 11
4.3 Managing the analyses 11
4.4 Creating a new analysis 11
4.5 Analyzing the algorithm 12
4.5.1 Extrapolations 14

5 Running Time Predictions **15**
5.1 Support of Block Sampling in MaLiJAn 15

6 Typical Problems, Known Pitfalls & Neat Tricks **16**
6.1 Costs 16
6.1.1 Splitting Costs 16
6.2 Counter Extrapolations 16
6.2.1 Input Size Range Hints 16
6.2.2 The Sum-of-$\mathcal{O}$-Criterion 16
6.2.3 Overfitted extrapolations 17
6.2.4 Customizing the Extrapolation Heuristic 17
6.3 Elimination 17
6.4 Analysis Result 18
6.4.1 Block-Counter-Extrapolation Integrity Check 19

7 Advanced Techniques **20**
7.1 Distributed Profiling 20
7.1.1 Slave Configuration 20
7.1.2 Client Configuration 21
7.2 Securing Slaves 21
7.3 Better Estimates of Free Memory 22

References **23**

1 Overview

MaLiJAn is a tool that analyzes an algorithm by training a stochastic model of the algorithm on user-provided input data and obtains its average complexity using maximum likelihood analysis (see section 2).

The rest of this manual is organized as follows: Section 1.1 introduces main steps of the analysis explained in further detail in subsequent sections. Prerequisites to be met by the algorithm are stated in section 3. Section 4 guides you through the analysis of your algorithm in MaLiJAn. Problems we encountered in analyses are discussed in section 6 together with "best practice" hints to avoid them. Section 7 explains setup for large-scale distributed profiling.

1.1 Steps of Analysis — A Glossary

The analysis consists of the following steps. Figure 1 shows the connections between these steps.

Augmentation The Java input algorithm is marked to produce costs on critical positions in the source code, then compiled and augmented with counters on basic block entry points that count how often they are used during program execution. The user specifies which program parts are augmented by manual selection and code commenting.

Profiling The augmented program is run on a given sets of inputs. Obtaining transition counters necessary to train the stochastic control flow model can take a while, especially if the algorithm under consideration is slow and/or some input data is large.

Therefore, MaLiJAn is able to distribute simulation to a cluster of machines using slaves connected by RMI as describe in section 7.

Analysis Once Profiling is done, in the analysis phase data is extrapolated and expected values for costs measures are computed. This phase consists of multiple tasks that determine the quality of the analysis result and may be iterated multiple times with adjusted parameters.

Control Flow Graph The control flow graph (CFG) of a program is a directed graph consisting of one node per instruction in the code. Two nodes u and v are connected by a transition edge (u, v) if there exists a possible run of the program in which v is executed *directly* after u.

As most instructions in a program are *sequential*, i. e. execution always continues with subsequent instruction, CFGs tend to contain linear lists. For reasons of clarity and comprehensibility, MaLiJAn contracts such lists into single nodes called *basic blocks*.

MaLiJAn displays the graph as well as detailed information about its basic blocks, which can be valuable help for identifying problems in analyses.

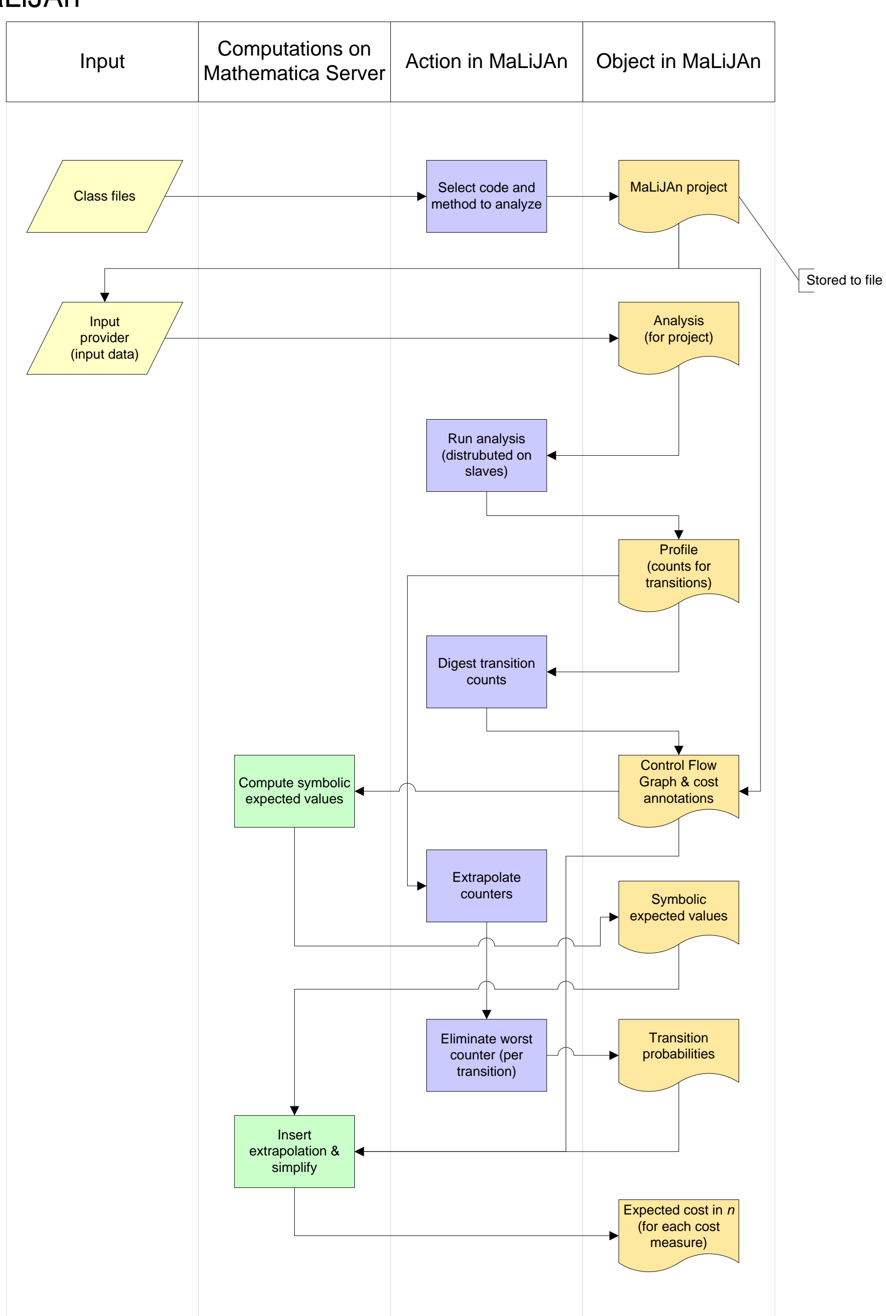

Figure 1: Overview flow chart of the different steps in MaLiJAn.

Compute Expectation The control flow graph of the algorithm is translated into a probability generating function (PGF). The PGF is used to obtain symbolic expected values of all cost measures in terms of symbolic transition probabilities (further details in section 2).

Extrapolation For each transition, collected data for different input sizes is extrapolated using a heuristic to get transition probabilities as a function in input size n.

This is done by extrapolating counters — how often blocks were entered and transitions were taken respectively — and then dividing extrapolations. Details in section 2.

PlugIn These transition probabilities are inserted in symbolic expected values for cost measures, resulting in approximations of average case complexities of the algorithm in terms of input size n.

Result Simplification Often, resulting expressions need to be simplified to draw conclusions from them. To this end, several simplification methods are offered.

2 ⚠ Primer — Theorectical Foundations

This section gives a brief introduction to the theory underlying MaLiJAn.

We assume basic familiarity with ordinary generating functions and their application to derive expected values from probability generating functions for discrete random variables with range $\mathbb{N}$. A nice introduction can be found in [1, Chapter 8.3].

2.1 Maximum Likelihood Analysis

Maximum **Li**kelihood **An**alysis was introduced in [2] as a tool to derive symbolic average case analyses of algorithms for empirical input distributions. The result is a symbolic expression for the expected value of some cost measure depending on the input size parameter n. MaLiJAn allows to automate large parts of this method and supports arbitrary user-provided algorithms (given as Java Bytecode).

Using information collected by running an algorithm on *user provided* input data — potentially taken from real-world applications and thus reflecting the actual distribution of inputs in practice — MaLiJAn can semi-automatically derive the symbolic average case costs of the algorithm. As done in classic runtime analyses, costs are defined in terms of *elementary operations*, e. g. “key comparisons” for sorting.[1]

The underlying idea is to regard the control flow graph of an algorithm as finite Markov chain: We abstract from the exact behavior of the program by only looking at the program counter, i. e. states are certain positions in the code of the program and transitions jump from one position to another. This model is completely described by considering conditional probabilities

$$p_{i\to j} = \Pr\left[\text{continue with basic block } j \mid \text{currently in basic block } i\right] .$$

In MaLiJAn, $p_{i\to j}$ will be rendered as $\mathsf{p(i,j)}$ for better legibility in small fonts.

However, it turns out this abstraction was too eager to still allow interesting conclusions: For most algorithms, $p_{i\to j}$ is *not constant* with respect to the input size n.

Therefore, we allow transition probabilties to *depend on input size* n

$$p_{i\to j} = p_{i\to j}(n) .$$

Theorem 5 in [2] now states that we can algebraically compute the average case runtime as follows:

1. Solve the symbolic linear equation system

$$B_i = \begin{cases} 1 & \text{if block } i \text{ is a sink} \\ \sum_j p_{i\to j} \cdot y^{k_j} & \text{otherwise} \end{cases} \qquad \text{for all basic blocks } i$$

   for the entry point B_0 of the main procedure. k_j is the cost contribution of basic block j (e. g. number of elementary operations).

[1] In MaLiJAn, “elementary operations” are called cost measures.
They are defined by the user (see Section 3.1).

The result is a rational function in variable z that contains *symbolic* transition probabilities $p_{i \to j}$. This function is the ordinary probability generating function

$$S(z) = \sum_{k \geq 0} \Pr[k \text{ elementary operations}] \cdot z^k$$

2. Compute $S'(1)$, the first derivative of S with respect to z at point $z = 1$.

   The result is the symbolic expected value for the number of elementary operations.

3. Replace symbolic $p_{i \to j}$ by $p_{i \to j}(n)$.

   The resulting term is the expected number of elemenary operations executed for input size n.

2.2 The Art of Extrapolation: How to get $p_{i \to j}(n)$?

In general, it is not possible to determine the functions $p_{i \to j}(n)$ automatically.[2]

However, for many cases the following heuristic works very well: We count in $c^{(n)}_{i \to j}$ for each basic block transition $i \to j$ *and* input size n how often it occurred in the execution of the algorithm. From them, we can easily compute $c_i := \sum_j c_{i \to j}$. Assuming one fixed input size n, $\frac{c_{i \to j}}{c_i}$ is the maximum likelihood estimate $p^{(n)}_{i \to j}$ of $p_{i \to j}$ at size n.

Hence, we can use extrapolations $\bar{c}_{i \to j}(n)$ and $\bar{c}_i(n)$ derived from the points $c^{(n)}_{i \to j}$ and $c^{(n)}_i$, respectively, to compute an extrapolation

$$\bar{p}_{i \to j} := \frac{\bar{c}_{i \to j}(n)}{\bar{c}_i(n)}$$

for $p_{i \to j}$.

Of course, it would also be possible to directly extrapolate the points $p^{(n)}_{i \to j}$. However, the extrapolation heuristic currently implemented in MaLiJAn are intended for monotonically nondecreasing functions — which is typical behavior for *counters*.

Additionally, experience with Maximum Likelihood Analysis has shown, that judging the quality of extrapolations — which is the only part of the analysis that has to be done with human guidance — is much harder for direct extrapolations of probabilities compared to extrapolations of counters. The apparent reason is that probabilties are naturally constrained to the interval $[0, 1]$ such that wrong extrapolations can still appear close to the points $p^{(n)}_{i \to j}$.

[2] At the very least, there are *uncountably* many of them ...

3 Prepare Your Algorithm

In order for MaLiJAn to analyze a Java program, you have to specify the following metadata:

set of analyzed methods Only code from those methods will be included in the control flow graph. This allows to focus on essential parts of an algorithm.

start method Among the set of analyzed methods, one method has to be selected as main method. This method will be called by MaLiJAn during profiling. The start method needs to be declared `public` and `static` and has to take exactly one parameter, say of type T. The inputs the algorithm is run on are instances of this type T.

cost measures Cost measures determine to what extend each code segment contributes to total costs. You may define several independent cost measures.

input provider The input data created by input provider is passed to the start method during profiling. Hence, it has to return inputs of type T.

3.1 In-code cost measures

MaLiJAn allows two different ways to specify cost measures. The probably most convenient one is to directly annotate costly instructions in source code.

To do so, simply add a call to (static) method `produceCost` in class `de.unikl.cs.agak.malijan.annotation.Annotations`:

```
produceCost("<cost-measure>",<amount>);
```

This assigns the following statement `<amount>` units of cost for `<cost-measure>`.

⚠ Method `produceCost` is designed to always returns `true`, so you can use it in boolean expressions:

```
if ( produceCost("key comparison",1) && a[i] <= a[j] ) ...
```

3.2 Compiling

Augmenting is done on Java Bytecode, therefore you have to provide your Java program in *compiled* form. Which Java compiler you use does not matter.

⚠ Including debug symbols on compilation makes basic block info in the control flow graph view much more informative. If you are using Oracle JDK, use parameter `-g`.

3.3 Unsupported Language Constructs

Some advanced features of the Java Programming Language have not been considered in MaLiJAn.

Exception Handlers are not included in control flow graphs. Example:

```
try {
    A: throw new Exception();
} catch (Exception e) {
    B:
}
```

Here, the "correct" control flow graph contains an edge from `A` to `B`, as the control flow continues in the exception handler. This edge (together with the node for `B`) is *not* created by MaLiJAn.

Multithreading For concurrent programs, it is in general not clear how to sensibly combine the profiles of different threads. Therefore, MaLiJAn currently does *not support multiple profiles at all.*

⚠ In the current implementation, if analyzed methods are executed concurrently, global transition counter will falsely "detect" transitions between basic blocks executed from different threads. The resulting CFG will contain incorrect edges and transition counters are just wrong.

Bottom line: Don't do it.

4 Analyze Your Algorithm

In this section we want to explain briefly how to use MaLiJAn in order to analyze a given algorithm.

Before we start with the walkthrough we have to introduce some GUI related terms:

Project: A project consists of a given *source base*, i. e. a folder or JAR-archive containing (compiled) byte code of a Java program to analyze. It further contains information about the selected methods to be analyzed as well as chosen start method (cf. section 3). Conceptionally, the project is divided in several *analyses* (see below). Every project is stored in a JAR-archive containing (possibly augmented) source base and a separate `*.maliconf` file (containing project's meta data). Both files must be kept in the same directory and must have same file name (of course with file extension `.maliconf` instead of `.jar`).

Analysis: An analysis is — as already mentioned — part of a project. In addition to information stored in a project, it further contains

- the chosen *input provider*,
- the *profile* containing all data collected during executions of the algorithm,
- as well as extrapolations and further formulæ created during analysis (see below).

The ordering of the following subsections corresponds to the workflow of MaLiJAn. First of all, when starting MaLiJAn, one has to either load an existing project or create a new one. We will start with the latter option.

Note that MaLiJAn allows only one open project at a time. However, you can start several instances in case you need to compare several projects.

4.1 Creating a new project

In a first step one has to specify a name for the project (which will determine the file name of the project files) and a source base. The source base can be a folder containing `.class` files or a JAR-archive. After pressing the 'Analyze' button it will be read and in case of success a tree of all classes contained in the source base will be shown.

Use checkboxes in the tree to select all methods to be included in the analysis. Only methods selected here can contribute to analysed costs of the algorithm. Hence, the rationale is to choose a *superset* of all methods possibly needed.

In the next step, the start method of the algorithm has to be chosen. MaLiJAn will only offer methods meeting requirements for start methods (`public`, `static` and one parameter; see Section 3).

After that the new project will be built, i. e. methods chosen to be analyzed are augmented for profiling and put into a JAR-archive named like the project without special characters. Additionally, the corresponding `maliconf` file containing already collected metadata will be created.

As from now, the project can be loaded as described in the next subsection.

4.2 Loading a project

If you want to open an already existing project, MaLiJAn offers the option to read a `maliconf` file. It will be checked, if the `maliconf` file contains valid data and if a corresponding JAR-archive can be found (needs to be located in the same directory) and opened. In case of success the *analysis management* will be shown.

4.3 Managing the analyses

After loading or creating a new project, MaLiJAn shows every previously opened analysis as a *tab* and an additional tab for managing analyses — i. e. investigating, deleting, opening or creating new analyses.

In the upper area of the program window several buttons are located. We explain them from left to right:

- Return to the start screen (thereby closing current project)
- Analysis Management & Analysis Tabs
  This is the main working area for doing analyses and the default view after opening a project.
- Project details.
  Shows a verbose text view of the open project and all its analyses.
- Status & History of requests (see below)
- Save current project.
  Note that MaLiJAn never saves your project automatically, except when building the project. (Reason is that we do not provide an undo button . . .)
- About dialog, giving information of MaLiJAn's origin.

4.4 Creating a new analysis

By opening a new tab one has the possibility to create a new analysis.

> ⚠ If other analyses have been created and profiled already, MaLiJAn will offer re-use their profiles and merge them with a newly created profile.
>
> This will come in handy if you notice during analysis that in the current analysis too little profiling data was collected. Then you can copy the existing profile and *add* e. g. some samples for larger input sizes.

In the next step one has to give a name for the analysis and select an input provider. If there are non-`abstract` classes implementing the `InputProvider`-interface in the underlying source base those will be shown. Additionally, MaLiJAn offers several built-in input providers. Note that only those input provider will be shown which input type is a subtype of the start method's argument type.

After selection of an appropriate input provider constructor there will be the possibility to enter values for the constructor's arguments.

If MaLiJAn is able to instantiate the chosen input provider with the chosen constructor and the entered arguments, one can continue with the profiling step. The input provider will be called as long as it returns new data, which will then be used to run the algorithm via the chosen start method. During execution all occurring basic block transitions will be counted. The profiling step can be aborted at any time without loss of the already finished results, and can be restarted if necessary.

> ⚠ See section 7 for how to set up a cluster environment to distribute profiling to several machines.

4.5 Analyzing the algorithm

As soon as profile data is set one is able to analyze the algorithm's cost. MaLiJAn offers a two-part view.

In the upper area of the screen, the program's (compactified) control flow graph is shown. Vertices correspond to basic blocks and an edges between two nodes indicates a *possible* transition from one block to the other.

The lower area holds several tabs for the different analysis steps In the following, we describe the typical workflow of an analysis (from left to right):

Overview: In this tab, some information about the underlying profile is given. Besides a plot of the number of runs per input size, there a button allows to export profile information.

Block Info: Here, detailed information is for the control flow graph is shown. When selecting one or more basic blocks, information about the corresponding byte code is displayed.

Cost Measures: In this tab, you define the cost measures to apply. Initially, only measures derived from code annotations are shown, but you may also create new cost measures from scratch and edit existing ones.

Once all cost measures are defined, click on 'Compute Symbolic Expectation'. Note that you cannot change cost measures after that.

Base Functions: This tab shows the list of base functions used for the extrapolation heuristic. For the analysis to be correct, you have to give (a superset of) the functions actually occurring for counters in this list. As those functions are not known up front, add all sensible ones here and use the next tab to evaluate their adequacy.

Currently, the heuristic is only intended for monotonically non-decreasing functions. Other base functions will produce results, but those might be poor extrapolations. Moreover, the list has to be sorted by descending asymptotic growth, which can be done automatically by clicking 'Sort'. If the sort method cannot determine for two functions f and g, which has the larger asymptotic growth rate, the user will be interactively queried.

Counter Extrapolation: This is one of the two extrapolation tabs. Here one can extrapolate the counters for basic blocks and transitions (see Section 2.2).

By selecting one or more entries in the list to the left, you can view plots of the counters. If the entry has already been extrapolated, the function is shown and included in the plot.

A click on 'Extrapolate' opens the Extrapolation Dialog, where you can adjust parameters for extrapolation. The dialog has two tabs, one for MaLiJAn's extrapolation heuristic and one to enter manual functions.

For the heuristic, you can select a list of *base functions*. The resulting function will always be a linear combination of those. Additionally, you can select an interval of input sizes. Only counters whose x-value falls in this interval are used for the extrapolation. (This allows exclusion of atypical points, see Section 6.2.1).

In the second tab of the Extrapolation Dialog you can enter a manual function term. It has to be given Mathematica syntax, where `n` is the input size.

Colors in the list of 'extrapolatable' entries indicate their states: A red entry has not been extrapolated yet. Once it was extrapolated, it is show in black. Gray entries are counters that do not occur in any symbolic expected cost value and hence need not be extrapolated.[3]

From Counters to Probabilities: After creating counter extrapolations, the next step is to compute *probability* extrapolations from those. For that, you have to select for each basic block which counter extrapolation to eliminate (the probability extrapolations are over-determined).

MaLiJAn can automatically select the "worst" extrapolation for elimination according to some built-in heuristics.[4] Hitting button 'Eliminate and Compute Probabilities', computes all probability extrapolations for this basic block. Any existing extrapolations will be overwritten.

Expected Costs: This is the last panel and offers functionality to complete the analysis. For each cost measure one can compute the expected value now.

After choosing a cost measure, the symbolic expected value will be displayed and you can plug in the extrapolated probabilities. As a result you get a single term in input size n for the expected value of the selected cost measure. Additionally, a plot of this term will be shown together with cost amounts determined during profiling to judge analysis quality.

In the combobox, MaLiJAn offers a variety of term simplification methods. Clicking on 'Simplify' applies the currently selected one to the plugged in term. You can use several simplification steps one after another.

[3] Of course, this is only known once you have clicked 'Compute Symbolic Expectation' in tab Cost Measures.

[4] As indicated by the term 'heuristic', these automatic methods may fail. You should always bear in mind Section 6.3.

Runtime Prediction: This tab gives access to the runtime prediction facilities of MaLiJAn. See Section 5 for a detailed description.

4.5.1 Extrapolations

To ensure a meaningful result, extrapolation quality needs to be checked. Extrapolations depend on the choice of base functions as well as the choice which counts should be included and excluded from extrapolating, respectively.

Since counts of the outgoing transitions of a base block sum up to the count of the block, the same has to hold for extrapolations. So one of those values is expressed in terms of the others instead of being extrapolated independently.

5 Running Time Predictions

Apart from the purely combinatorial cost measures discussed in the last section, MaLiJAn can also be used to predict actual running times on a particular machine.

In principle, it suffices to measure the running times of all basic blocks. However, those running times are in the range of nanoseconds, so they cannot be properly measured directly. MaLiJAn contains an indirect methodology called "basic block sampling" to determine these block running times. It is fully automatic and aims at minimizing systematic errors. We divide the program into basic blocks, i. e. maximal blocks of sequential instructions. Then, we inject instructions at the beginning of each block to store an identifying number of the block in a global variable. This introduces a systematic error as each basic block becomes a few instructions longer, but it will be fairly small compared to other techniques of runtime measurement. Then, on a periodic basis, we concurrently read the global variable and store the block number. Note that this periodic job is done in parallel and hence does not influence the running time of the algorithm itself, i. e. it does not add to the systematic error. By repeating the run sufficiently often, the relative frequencies of the observed block numbers approach the relative running time contribution of the blocks.

From this, we get the vector $b = (b_1, \ldots, b_k)$ of observed block frequencies, i. e. block i has been seen b_i times in total. In separate runs, we also count f_i exactly, i.e. how often block i is executed in total, and we measure the total running time T in yet another run. Then, we use

$$c(i) := \frac{1}{f_i} \cdot \frac{b_i}{B} \cdot T, \qquad \text{where } B := \sum_{i=1}^{k} b_i$$

as an estimate of the block running times.

5.1 Support of Block Sampling in MaLiJAn

Unfortunately, MaLiJAn currently only gives preliminary support for block sampling.

In the 'Runtime Prediction' tab of an Analysis, there are two buttons, 'Sample' and 'Measure'. Those create an standalone executable JAR archive, which can be started on the target machine to automatically measure b and T using the inputs created by the Analysis' InputProvider.

Computation of the block costs $c(i)$ and integration into MaLiJAn is planned for a future version. For the time being, one has to do the processing externally.

6 Typical Problems, Known Pitfalls & Neat Tricks

6.1 Costs

6.1.1 ⚠ Splitting Costs

If the expected value for some cost measure is too complicated to handle, you might try to split up cost measures into several ones. This works if the cost measure assigns non-zero amounts to *several* basic blocks:

Let $B_1, \ldots, B_k$ be the blocks, where cost measure C has non-zero value. The you can define k simpler cost measures $C_{B_1}, \ldots, C_{B_k}$, where C_{B_j} only assigns cost to B_j.

Since expected values are linear, we have

$$\mathbb{E}\, C = \mathbb{E}\Big(\sum_j C_{B_j}\Big) = \sum_j \mathbb{E}\, C_{B_j}$$

.

6.2 Counter Extrapolations

6.2.1 Input Size Range Hints

The selection of input sizes to use for extrapolating is vital for two reasons:

1. Judging quality from plots is only possible by looking at points that were *not* used for computing extrapolations. As we mainly need to hit the correct *asymptotic growth* of functions, some of the largest profiled input sizes should be excluded from extrapolating.
2. Many algorithms show their typical runtime behavior only for inputs exceeding a certain size. Below that size, initialization code etc. might hide actual hot spots. Therefore, excluding small sizes can help if they do not fit the global trend.

6.2.2 ⚠ The Sum-of-$\mathcal{O}$-Criterion

How to judge the quality of an extrapolations in general is not clear. Before we resort to guessing, necessary conditions for sensible extrapolations should be checked. One example is the **Sum-of-$\mathcal{O}$-Criterion**: For arbitrary functions f, g holds:

$$\mathcal{O}(f+g) = \mathcal{O}(\max(f,g)) = \begin{cases} \mathcal{O}(f) & \text{if } g = \mathcal{O}(f) \\ \mathcal{O}(g) & \text{otherwise} \end{cases} .$$

For the counter extrapolations, we always have the situation that the sum of all transition counters $c_{i \to j}$ equals the block counter c_i. Hence, in terms of $\mathcal{O}$, a sensible set of extrapolations fulfills

- $c_{i \to j} = \mathcal{O}(c_i)$ for all target blocks j.
- $c_{i \to j} = \Theta(c_i)$ for at least one block j.

This criterion is automatically checked by MaLiJAn's elimination heuristic, so if the heuristic did not fail, the Sum-of-$\mathcal{O}$-Criterion is guaranteed to be fulfilled.

6.2.3 ⚠ Overfitted extrapolations

The extrapolation heuristic always yields a linear combination of the base functions (plus some constant). The maximal number of base functions with non-zero constant factor can be given in extrapolation dialog. Experience suggests that rarely more than 2 should be used.

If the heuristic does not find a suitable extrapolation with one base function, often too few or too noisy data is used, such that further profiling might help. Using several base function terms often looks well in plots, but does not hit ground truth.

Of course, the correct function might also be completely missing from the list of base functions. Then, you will have to add it to the set of base functions.

6.2.4 Customizing the Extrapolation Heuristic

If the extrapolation routine selects a function that does not seem right, you might try the following steps

1. Change the set of selected points, see section 6.2.1.
2. *Deselect* the leading term of the current extrapolation from base functions. By that, the heuristic can be prevented from using seemingly wrong term again.
3. If you have reason to believe a certain *base function* should be correct, select *only* this function as base function. The result will then be of the form $a \cdot f(n) + b$, where $f(n)$ is the selected base function.
4. ⚠ If your concrete guess is not part of the set of base functions, you can change the list on the 'Base Functions' tab. Function terms have to be given in Mathematica syntax

   Important: Currently, only monotonically *non-decreasing* base functions are supported and the list of base functions has to be sorted by asymptotic growth. The latter can be done automatically using the 'sort' button. The heuristic will produce *some* result even for base functions violating these conditions, but the extrapolations might be completely wrong.
5. ⚠ If the built-in heuristic completely fails, you might use the CSV-export of the points and come up with an extrapolation manually. Export is available in 'Counter Extrapolation' tab via button 'Export Counters'.

   Once you found a suitable extrapolation, enter the function term in the Manual Term tab of the extrapolation dialog.

6.3 Elimination

After all counters have been extrapolated, we select "the worst" of them and throw it away. The eliminated extrapolation can be expressed by the others using $c_i = \sum_j c_{i \to j}$.

Before we go into detail of elimination, let a warning be issued:

If a counter extrapolation looks wrong,
do not try to solve this by eliminating it!

Elimination of counters can help reduce numerical errors,
but in general it cannot correct for wrong $\mathcal{O}$-classes in extrapolations.

Which counter is eliminated can heavily influence the result of the analysis (for reasons given below). In most cases, you will get best results by the following procedure, which is implemented in MaLiJAn's heuristic for automatically selecting a counter:

1. Find all transition counts with the same $\mathcal{O}$-class as the block count. As argued in section 6.2.2, we find at least one such (or some extrapolations are wrong).
2. Eliminate either the block count or one of the found transition counts.

   ⚠ Potential numerical errors should guide your decision: Higher total count values are typically more stable than lowers; extrapolations of points that form a smooth line will be more reliable than ones resulting from fuzzy point clouds.

To motivate this procedure, consider a basic block i, with two outgoing transitions $i \to j$ and $i \to \ell$, and assume the extrapolations $\bar{c}_i$, $\bar{c}_{i\to j}$ and $\bar{c}_{i\to\ell}$, respectively, fulfil

$$c_{i\to j} = \Theta(c_i) \qquad \text{and} \qquad c_{i\to\ell} = o(c_i)\ .$$

If we selected $i \to \ell$ for elimination — contradicting above procedure — we would replace $\bar{c}_{i\to\ell}$ by $\bar{c}'_{i\to\ell} = c_i - c_{i\to j}$. As the extrapolations are computed numerically, it is probable that the coefficients of the leading terms in c_i and $c_{i\to j}$ are not equal and then

$$\bar{c}'_{i\to\ell} = c_i - c_{i\to j} = \Theta(c_i)$$

which is certainly wrong.

6.4 Analysis Result

After plugging in probability extrapolations, the final expression for the expected value can be rather huge. MaLiJAn provides several general-purpose methods to simplify those — however a reliable *and* tractable way of condensing the term down to its dominant contributions is still part of active research.

To our experience, the following sequence of simplification steps work in the vast majority of all cases:

1. 'Asymptotics via Base Functions' to get rid of terms vanishing for $n \to \infty$.
2. 'Convert to Numeric' to convert (rational) constants to decimal fractions.

For the remaining cases, you will have to confine yourself to trial-and-error. Each simplification is applied to the result of the previous one, so you can apply several of them in a row. Hitting the plug-in-button resets the expression.

> ⚠ A word of caution concerning the plot 'Computed Expectation vs. Sampled Costs' is in order:
>
> Even small numerical errors in the counter extrapolations might pile up to a significant difference in the resulting expected value. A rather bad looking plot might still hit the correct $\mathcal{O}$-class of growth.
>
> On the other hand, of course, a plot that looks well is not sufficient for a correct analysis.

6.4.1 ⚠ Block-Counter-Extrapolation Integrity Check

A simple cross-check for counter extrapolations and final results is based on the following fact: Assuming the block counter extrapolations are correct, we can compute the expected value of a cost measure C as

$$(\mathbb{E}\, C)(n) = \sum_{i \in \mathit{Blocks}} C(i) \cdot \bar{c}_i(n)$$

if we interpret C as a function $C : \mathit{Blocks} \to \mathbb{N}$ assigning to basic block i the amount $C(i)$ it contributes to the cost measure C.

Using this identity, we have a second way to determine expected costs, yielding a way to check integrity of the analysis.

7 ⚠ Advanced Techniques

7.1 Distributed Profiling

MaLiJAn is ready to use an arbitrary numbers of remote machines to profile faster. As explained above, MaLiJAn starts one subprocess per input that runs the algorithm on that particular input. This paradigm beautifully scales to many machines. Any number of workers—we call them *slaves*—can request individual inputs from clients and profile them independently.

To that end, you need to have a set of computers your office machine can connect to and vice versa; ideally, they are all situated in a local network with no NAT, firewalls or similar ailments involved. On each of your slave machines, you configure and run our slave application which we call malijan-slave from now on. The client you use needs some configuration, too. This section will teach you how to do both things so you can profile as speedy as a desert mouse runs.

7.1.1 Slave Configuration

After you download and extract MaLiJAn slave you have to set up a couple of parameters for a functioning slave. Put a file named <username>.properties (replace <username> with your username) next to malijan-slave.jar you downloaded. A property is specified by putting <property-name> = <value> in a separate line. Lines starting with # are ignored. You have to specify at least these parameters properly:

host The address clients can reach the slave under. This can be an IP or a proper alias, e.g. a hostname that is resolved by a DNS your clients have access to. If you malconfigure this property clients might still be able to connect initially but not send you any profiling requests.

> ⚠ This property sets Java's system property java.rmi.server.hostname which is used as callback address in remote objects passed to clients.

port, objectport These two parameters are actually optional as the defaults 1099 and 1337, respectively, should work well in most circumstances. Make sure clients can connect to whatever ports you end up using.

> ⚠ port is used for the slave's RMI registry while objectport is the port remote objects listen on.

> ⚠ For a full parameter list with comprehensive descriptions, check slave-default.properties in malijan-slave.jar. Among others, you can change the slave's name, how much resources it may use and where to write temporary files to.

Now you are almost done. All you have to do is open up a terminal in your slave directory and run malijan-slave start[5].

[5] You may have to make the script executable: chmod +x malijan-slave

⚠ We currently only support slaves running on Linux via our runscript. You can, of course, start slaves by hand on any machine with `java -jar malijan-slave` provided that you manually put a file named `malijan-slave.pid` in your system's default temp folder.

Check the log file appearing in the same directory; it should contain a message saying that your slave is working for work now. If not, something went wrong and the log hopefully contains information to help you — or us — to fix the problem.

⚠ In order to follow logging messages more closely, run `tail -f` on the logfile. You can adjust logging granularity by defining property `loglevel`; valid values are `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL` in order of decreasing verbosity.

7.1.2 Client Configuration

For MaLiJAn to use remote slaves for profiling, you have to set the following properties in `<username>.properties` (replace `<username>` with your username) next to `maliclient.jar` you downloaded. A property is specified by putting `name=value` in a separate line. Lines starting with # are ignored.

slaves A comma-separated list of slave addresses with port. Address and port should be the same as the respective slaves `host` and `port` properties, respectively.

host The address slaves can reach the client under. This can be an IP or a proper alias, e.g. a hostname that is resolved by a DNS your slaves have access to. If you malconfigure this property slaves will not be able to send their results back.

⚠ This property sets Java's system property `java.rmi.server.hostname` which is used as callback address in remote objects passed to slaves.

port This parameter is actually optional as the default 1337 should work well in most circumstances. Make sure slaves can connect to whatever port you end up using.

⚠ `port` serves the same purpose as `objectport` on slaves. If you run multiple clients or both client and slave on one machine, make sure to choose a different port for every instance.

7.2 ⚠ Securing Slaves

Slaves run with rather liberal security permissions[6] by default. In particular, they have read and write access to all files the executing user can access and accept remote connections from everywhere. This can be restricted by carefully chosen host-dependent permissions.

Check `slave.policy` in `malijan-slave.jar` to find the default permissions. Slaves need at least the following permissions to run as intended:

[6] Check http://download.oracle.com/javase/6/docs/technotes/guides/security/permissions.html for an introduction.

```
permission java.lang.RuntimePermission "modifyThread";
permission java.lang.RuntimePermission "shutdownHooks";
permission java.util.PropertyPermission "*", "read";
```

It is up to you to configure both `FilePermission` and `SocketPermission` properly. As for files, slaves will need read, write and delete access to the configured working directory and have to be able to execute the local Java installation. As for sockets, slaves will only be able to be used from clients that they are allowed to both accept connections from and to connect to.

Once you have come up with a policy file you can make your slave use it by setting property `policy` to its filename. Enable debug logging and check the log to see if everything works.

If you do not trust `malijan-slave` to honor your policy you can of course start it with a security manager of your choice by specifying the corresponding command line parameters. Good luck.

7.3 Better Estimates of Free Memory

If no suitable settings are given, MaLiJAn and `malijan-slave` try to use as much memory as possible for profiling. To that end, they check how much memory is free. The default estimates can be bad so we implemented an alternative in case you run into problems.

If you need better automatic estimates, download the SIGAR binary suitable for your system[7] and place it next to your client or slave run script.

[7] support.hyperic.com/display/SIGAR/Home

References

[1] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*, volume 2. Addison-Wesley Reading, MA, 1994.

[2] U. Laube and M.E. Nebel. Maximum likelihood analysis of algorithms and data structures. *Theoretical Computer Science*, 411(1):188–212, 2010.