Haskell in the Large

Jurriaan Hage

Technical Report UU-CS-2013-013
July 2013

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Haskell in the Large

Jurriaan Hage

J.Hage@uu.nl
Dept. of Computer Science, Utrecht University

Abstract. In this paper I describe my experiences during a few weeks of performance debugging (generated) Haskell and attribute grammar (AG) code. This paper has appeared non-peer-reviewed in the Liber Amicorum of Doaitse Swierstra [1], and has been taken over verbatim (except for this one sentence).

1 Introduction

Haskell may be suitable for writing small programs elegantly, but as a law of Software Engineering (DeRemer's, [1]) tells us: what applies in the small does not (always) apply in the large. I have found that to some extent this is still very much true for Haskell, and much work will need to be done to overcome these problems. This is also good news, because it will keep us off the streets for just a bit longer.

In this paper I describe some of the adventures I had (in March/April 2012) while performance debugging a large Haskell implementation of the Asic bytecode instrumenter for Shockwave Flash that was developed as part of the Fittest project (http://www.facebook.com/FITTESTproject). It is a somewhat incomplete and anecdotal story, because I tried many things, and did not keep an extensive diary. Indeed, much of what I write here is reconstructed from reading old e-mails, and looking at svn commits.

Although performance has improved to a sufficiently high level, and there is help to be found for this task (both from books like Real World Haskell [2] and the more recent implementations of the attribute grammar system [3]), I found that performance debugging is very much an art, and by no means simple. Moreover, when you consider applications that work at a scale such as this, you quickly get the feeling that you are the first to do so, running into issues that nobody you talk to has ever run into.

2 Performance debugging the Asic compiler

The Asic compiler is a bytecode instrumenter for the ActionScript language (which compiles to Shockwave Flash (.swf) files). It was developed largely by Arie Middelkoop when he served as a scientific programmer within the Fittest project. There is a paper about this work [4], but what follows was actually work done after that paper had been published.

2.1 The issue at hand

The bytecode instrumenter had been tested on some realistic Flash executables, such as the Flex-Store, but not on Sulake's Habbo Hotel (www.habbo.com) which is a much larger program, and which, at that time, was supposed to be our case study. In Shockwave Flash files, code, and other resources such as graphics, are organised into so called tags. A single code tag typically contains the implementation of one class. Therefore, code is usually spread over a large number of tags, each of them reasonably small. One reason why Habbo Hotel was so difficult to deal with, is that the people of Sulake had merged the code of numerous tags into just three tags, in order to obtain a smaller bytecode file.

[1] http://www.cs.uu.nl/people/jur/liberdoaitseswierstra.pdf

When compiled, the original Habbo has no more than 47,082 instructions in any of its tags, and only a small percentage goes over 10,000. The merged version we had to work with had a code tag with 1,479,958 instructions. Tags can be (are, in fact) transformed independently of each other. But our implementation was (and is) not so clever that it can divide a large tag into pieces and consider these pieces separately.

After Arie had left, we found that instrumenting the Habbo Hotel bytecode was way beyond our capabilities: a (sizable) instrumentation on the code led to memory consumption exceeding 32 GB. (We do not know how much it really was. All we know is that the program crashed on a 32 GB Linux machine.) Since we wanted an instrumentation of this kind to be performed on a "simple" MacBook with 4 GB memory, and we saw no reason why our instrumentation was supposed to take so much memory, I set myself the task of performance debugging the compiler. Since there was so much code, and pretty complicated code at that, this could only work effectively if I could tweak the performance by non-invasive, localized changes. This took some doing, but in the end, memory consumption was brought back back to 2.7 GB (saving quite a bit of running time as well). Below, I reconstruct (I have no exact records) some of what I, and others, did to attain this result.

2.2 The first few steps are easy

The first step is familiarisation with the application: there was a program called `asli` that depended on a library called `asil` (later renamed into `abci`). The library has 14,361 lines of hand-written Haskell and AG code, the program only contains 36 lines of code (in both cases, excluding empty lines and comments).

After looking at the top level functions, two things struck me. It seemed that somebody had tried to improve memory consumption by adding explicit strictness annotations here and there. However, when you parse a large file, putting a single strictness annotation (like ! or a seq) may not help much. What was needed here, in order to read in the whole file at once, was to deepseq the reading of the file. Second, there was a line of code that generated what amounted to a huge disassembly file of the bytecode. Although that may be useful for debugging purposes, this single line was the major reason for the huge memory consumption. After removing it, the program was down to 23 GB internal memory usage. This is a substantial improvement compared to not knowing at all what the actual memory consumption is. I also saw a reduction of memory consumption from 260 MB to 110 MB and running time from 60 to 40 seconds for the identify transformation on the smaller test program `test_apdf.swf`. After these easy wins the real work started.

3 The uphill slope

A trying part of this work was to have a suitable environment for running the (profiling) experiments. On the one hand I needed a machine with lots of memory, on the other I needed to be able to profile the code. This combination did not exist on any computer in our faculty accessible to me. In Haskell, for profiling to work, you need access to profiled versions of the base libraries. Unfortunately, the Linux server that enjoyed eight cores and 32 GB of internal memory did not have a GHC installed with the profiled base libraries. I asked system administration to install them, but when they finally told me they had done so, the work described here was already done. Instead, I chose a simpler route: first I verified that a similar memory problem arose for a smaller case study, and then I used that case study to see what the effects of changes were. Another advantage of this way of working that the transformations, when they succeeded, took much less time.

But how to profile Haskell code? I had no experience whatsoever at the time, but found the very helpful Chapter 25 of the book Real World Haskell [2]. The protocol described there strongly resembles my approach, so I will not go into it deeply; you can read it there. I also needed access to the list of flags that GHC comes with, because Real World Haskell does not give you the whole story [5].

Essentially, you typically need to run your program *multiple* times with different parameters for what the profiler needs to track for you. The first step is typically to start with the `hc` flag:

```
asli ../InjectionSpec.txt test_apdf.swf Hex +RTS -hb -s
```

runs the `asli` program, with a specified transformation (laid down in the file `../InjectionSpec.txt`, on the Shockwave Flash file `test_apdf.swf`, resulting in the transformed output `Hex`. The profiling is turned on by passing `-hb` and `-s` to the run time system, which is why they follow `+RTS`. The flag `hb` (b stands for *biographical*) says that you want to know, over time, what amount of memory is lag, use, drag and void. The latter describes memory that is never used (which may, for example, be due to some strictness flags being in the wrong place). *Drag* describes memory from its last use to its final garbage collection. These two types are the ones to look at first. The *lag* kind of memory is memory that is allocated long before its use. This can be problematic too, because the bottleneck of memory, and what I was trying to reduce here, is the maximum amount of memory in use across execution (reduction of memory *volume* is also nice, but that tends to follow the reduction of the maximum memory usage as a matter of course). *Use* is the time between the first and last use of a memory cell. The `hb` profile quickly gives you an idea of whether quick gains can be made. I used it too, but not overly much (discovering its use fairly late in the game).

The flags I employed the most were `hy`, `hc` and `hr`. The former informs you how much of your memory is occupied by each of the constructors in your program. Here, you may find out for example that lists taken an inordinate amount of memory, and if you happen to be working with long strings, you may want to switch to `Text`.[2]

The second important flag, `hc`, tells you which function created the need for a particular memory cell. The third flag, `hr`, tells you which part of the program *retains* how much of the memory. In this particular case, many of these were semantic functions generated by the AG compiler [3]. After consulting Jeroen Bransen and Arie Middelkoop, I learned about some recent innovations to the system, including a flag called `kennedywarren`. The purpose of this flag to decrease memory consumption by statically deciding on the order in which attributes are evaluated. This certainly helped, but not enough to attain my goal.

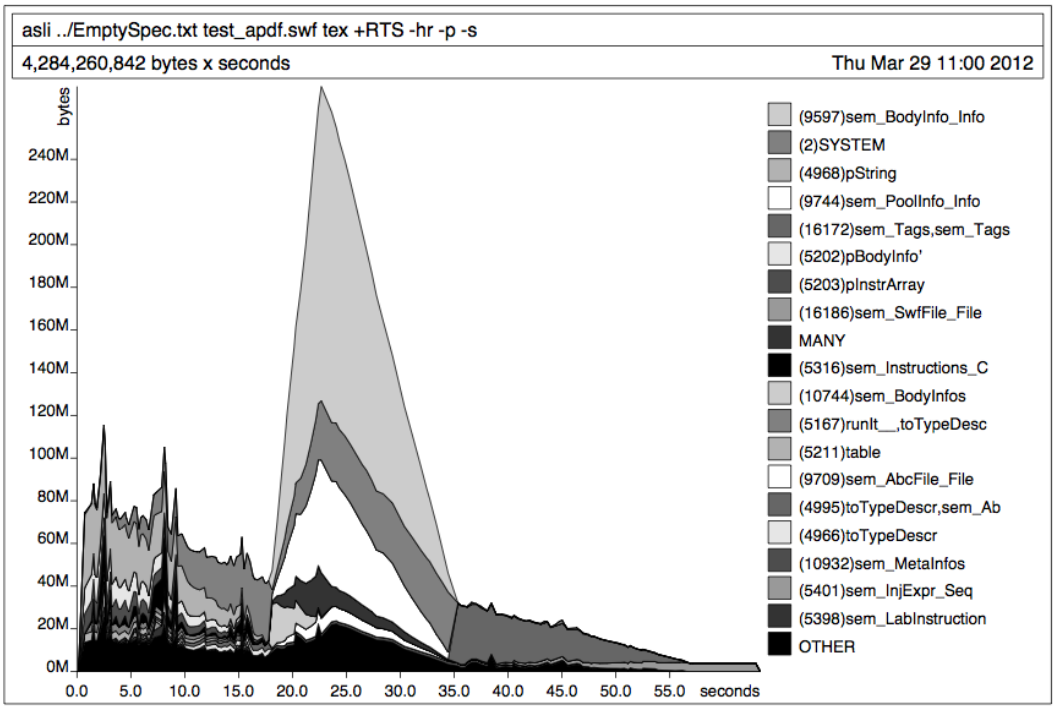
The progression of results is illustrated by the sequence of profiles made for an empty transformation (which reads and analyzes but does not transform the bytecode) in Figure 1 and 2. The full injection takes more time and memory, but generally follows this same pattern. In Figure 1(a) we can see a large spike between the 20 and 35 second mark. This is due to the fact that the disassembly file will be written out shortly. This spike is absent in the other two images. In all images we have depicted the retainer profile. In the legend on the side it can be seen that many retainers are functions starting with `sem_`; this implies that these are semantic functions generated by the attribute grammar system. There are many more such pictures to be displayed here. However, these three should be enough to convey the general flavour of these profile images. Note that these pictures are still very much in the early stages. Later, we got the memory consumption for a *full* instrumentation of this file below 100 MB.

3.1 The waiving of flags

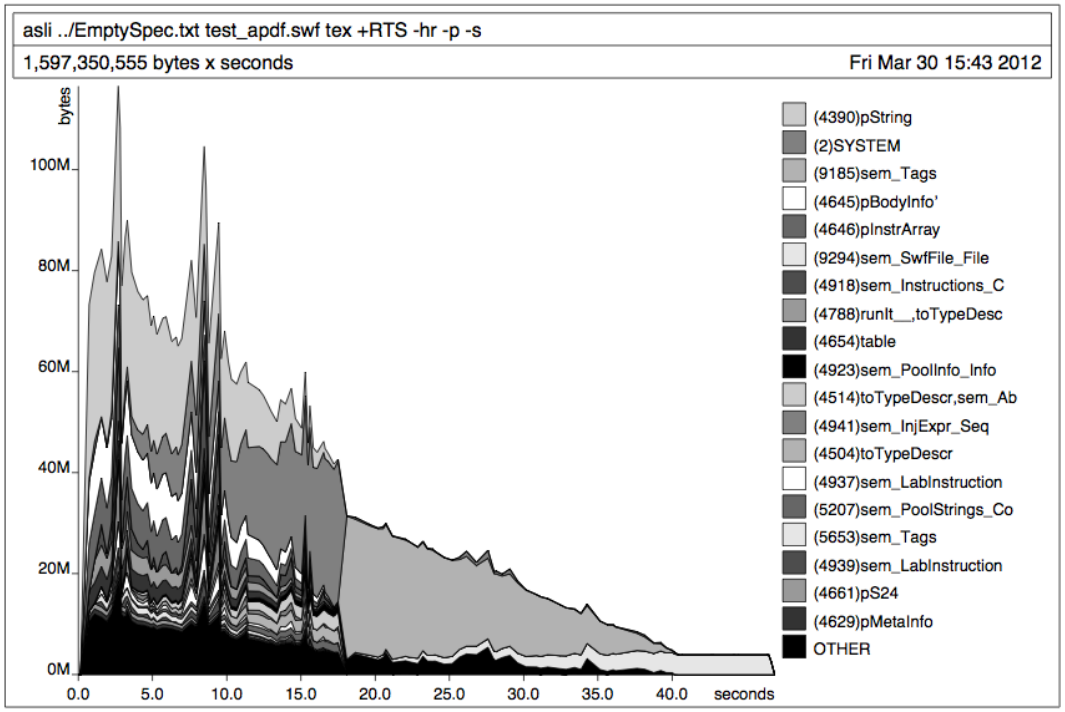
To close this gap, Arthur Baars and I sat down at the Utrecht Hackathon for an extensive profiling session. Here we found that what seemed to be the bottleneck for the computation: a small piece Haskell code that was generated from AG code, only twenty lines or so.

By hand, the code could be made to evaluate more strictly, thereby resolving the last of the memory bottlenecks. The problem was that I had to tweak the generated code, and had, as yet, no way of tweaking the AG code in a way that would have that effect on the generated code. Arthur Baars told me that he found some flags in the AG system that had the effect I was looking for.

[2] We did not need that here, because we worked with binary files. But in another performance debugging session (on the Haslog library that we also use within Fittest) we gained a huge win by switching to the `Attoparsec` parser library, and by switching from `String` to `Text`.

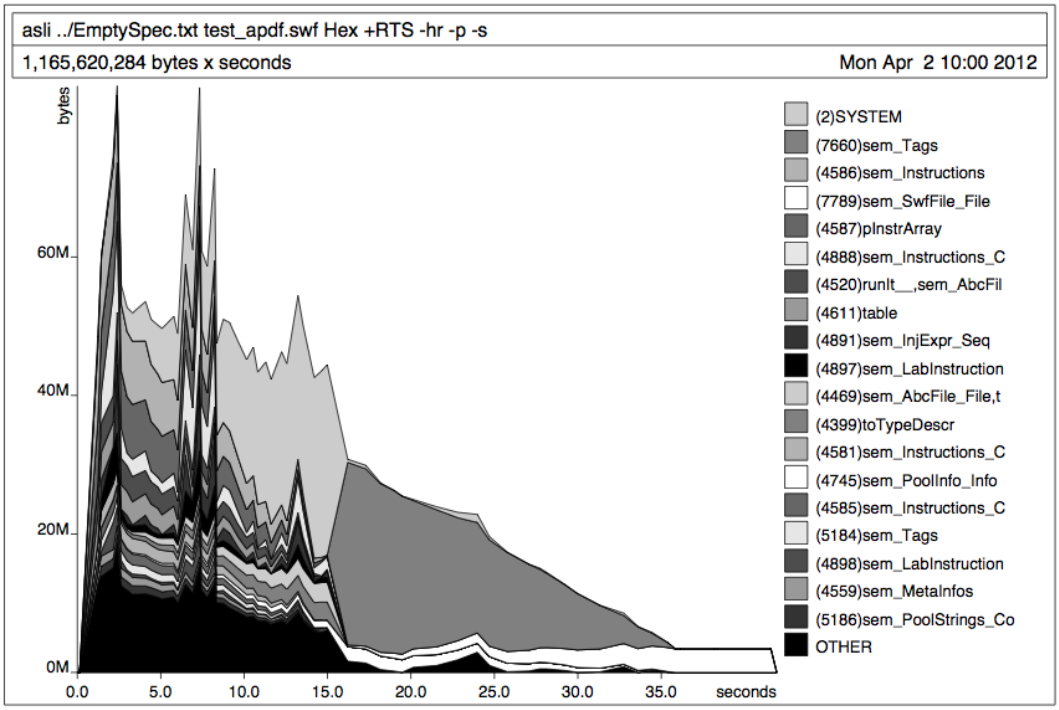


(a)



(b)

Fig. 1. The first and second profile



(a)

Fig. 2. The third profile

Indeed, after adding the `bangpats` and `optimize` to each of the AG modules, almost completely solved the remaining memory issues. Almost completely? Yes. Almost.

Maybe you have once tried to get the air out an inflatable mattress by putting yourself squarely on top if it. If you do not do so with care, often the effect is that the air is not forced out, but to a different part of the mattress. This is what seemed to happen here as well: a new bulge cropped up where there had not been one.

On a whim, I decided to omit the `bangpat` flag for the module that seemed to retain the memory in this spike, and so managed to get rid of this final memory hiccup in the profile. Running this version for the full injection on a particular version of the Habbo case study gave a maximum memory usage of 1.8 GB. Of course, at this point the application still uses quite a bit of memory, but at least the profile is now pretty much horizontal.

3.2 Tune in, tune out

Now that we seem to have the code under control, it was time for some fine-tuning. A lesson well-learnt is that if you have memory to spare, you can pass as argument `-H2000M` to the runtime, because then the program will "waste" much less time on garbage collection. In the case of full injection on habbo_secure, the running time goes down from 975 seconds to just over 300. Of course, the memory footprint increases, because the garbage collector is rarely called, and therefore do es not economically compact. This experiment was repeated on a Linux server that has many cores and much memory. The base timings for full injection without a `-K` option are 1526.72s When passing `-H4000M` to the execution, this becomes 658 sec. Surprisingly, when you choose `-H2000M` on this machine, the running time becomes smaller, 546s. So time is a function of the H option, but it is not monotonically decreasing with memory increase. It seems that anything between `1900M` and `2500M` gives comparable results in this case, and they get worse outside this interval.

Note that setting aside `4000M` for computation implies that when gc starts, the process will need quite a bit more than that. In fact, it used about double what I set. This implies that setting a

low value is better if that does not negatively affect run time too much. Using threads and parallel GC's only served to increase running time, sometimes by inordinately much. In some cases, I could do the transform faster by hand.

3.3 Some further considerations

Suppose you set yourself the task of bringing the running time of some application down to x milliseconds. It is then important to remember that the resource consumption of a program that has been compiled with profiling turned on is much higher than of a program for which this is not the case, *even if you do not ask the profiled program to actually deliver a profile*. This overhead is substantial, so it may be wise to, once in a while, compile a clean version without profiling and debug information to see what the situation really is like. According to the GHC user guide, compiling with profiling on gives code that has about 30 percent memory overhead.

About garbage collection, the GHC User Guide has the following to say [5]:

> Garbage collection requires more memory than the actual residency. The factor depends on the kind of garbage collection algorithm in use: a major GC in the standard generation copying collector will usually require 3L bytes of memory, where L is the amount of live data. This is because by default (see the +RTS -F option) we allow the old generation to grow to twice its size (2L) before collecting it, and we require additionally L bytes to copy the live data into. When using compacting collection (see the +RTS -c option), this is reduced to 2L, and can further be reduced by tweaking the -F option. Also add the size of the allocation area (currently a fixed 512Kb).

To this information I can add my own findings during this experiment that the overhead for the copying collector seems close to 2L, while the compacting garbage collector scores worse (both in terms of running time, and in terms of memory consumption) than the copying garbage collector. The former was no surprise, but the second came quite as a surprise to me. I also found during these experiments with the compacting collector that it sometimes simply crashed the application. I expect that this is because not many people still use it, and maintainance of this code is not a high priority.

It will interest at least one reader that we made good use of Doaitse Swierstra's error correcting parsers while building the bytecode instrumenter [3]. The reason is this: Arie quickly found that the bytecode specification and actual bytecode as it was understood by the Flash runtime diverged. In order to debug the specification (since we *had* to deal with actual bytecode generated by the ActionScript compilers), we employed the error correcting parsers to discover what the specs should have said, thereby obtaining a more precise (and even executable) specification of said specs.

When the Asic compiler was performance debugged, and we ran it on the Habbo Hotel testcase, we ran into another problem. Flash code needs to maintain an invariant on the size of the stack: for any given instruction, the height of the stack should always be the same when you execute that instruction, independent of how the statement is reached. To verify that our implementation did not break that invariant, Arie had implemented a dataflow analysis. It then turned out that after instrumentation Asic generated messages like:

```
param analysis at method 6 and instruction 726: warning: different
stack usages on incoming CFG paths: stack[1] { lbls: 703} : ...
and  stack[0] ...
```

The question was then: is our instrumenter wrong, or is something else the matter. After finding that running an identity transform on the case study revealed the same problem, I concluded that the problem was already in the original code. But why did the Flash runtime not scream bloody murder when we ran it? Then I remembered that Sulake performs extensive obfuscations on their code, merging tags, changing identifier names. Maybe the broken invariants are due to something they did to their code, but in a way that goes unnoticed when you execute it. With all these ingredients, the solution was simple: Sulake introduced *dead* code that when executed

would break the invariant. Our analysis was not precise enough to discover that the dead code was dead, which led to our failure to decide that the invariant was in fact not broken. To fix this, we simply allowed invariant checking to be turned off, because it was mostly useful in the development phases. A pleasant side effect is that this again reduced the running time of our instrumenter.

4 Closing words

It took me quite a bit of time to decide what to actually write about. I could have written about work Stefan, I and others have done on type and effect systems ([6, 7]), or about the topic of type error diagnosis that you initiated at Utrecht with the hiring of Bastiaan Heeren [8–12] that I am currently actively pursuing again. I could also have finished that (strongly typed) implementation of switching classes, seeing how the efficiency of counting the number of trees (or otherwise) [13–17] in a switching class would measure up against my C implementation (that it is faster than my earlier Scheme implementation can be considered a given). Instead, I opted for a topic that, I believe, is more to your heart, involving both Haskell and your own AG system.

Over the years, I have learned a lot from you. And although I was not always happy with your decisions, I have always found your loyalty and sense of duty towards the people in our group, to the department, and to science, admirable. I hope you enjoy your well-earned rest up there in the north, far away from people who put dead links on websites, who think that a lecturer need not sign for a passing grade of a student, or who believe that you publish papers, not results. As you well know, la vida en la ciudad es muy complicada. I just hope life in the village of Tynaarlo will not be too sedate.

Acknowledgements I gratefully acknowledge the assistance of Arie Middelkoop and Arthur Baars in the process of performance debugging.

References

1. DeRemer, F., Kron, H.: Programming in-the-large versus programming-in-the-small. In: Proc. Int. Conf. Reliable Software, IEEE Computer Society Press (1975) 114 – 121
2. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. 1st edn. O'Reilly Media, Inc. (2008)
3. Swierstra, S.D., Rodriguez, A., Middelkoop, A., Baars, A.I., et al., A.L.: The Haskell Utrecht Tools (hut) http://www.cs.uu.nl/wiki/HUT/WebHome.
4. Middelkoop, A., Elyasov, A.B., Prasetya, W.: Functional instrumentation of actionscript programs with asil. In Gill, A., Hage, J., eds.: Implementation and Application of Functional Languages - 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3-5, 2011, Revised Selected Papers. Volume 7257 of Lecture Notes in Computer Science., Springer (2012) 1–16
5. The GHC Team: The Glorious Glasgow Haskell Compilation system user's guide, version 7.0.1
6. Holdermans, S., Hage, J.: Making "stricterness" more relevant. Higher-Order and Symbolic Computation **23** (2011) 315–335
7. Holdermans, S., Hage, J.: Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In: Proceedings of the 15th ACM SIGPLAN 2010 International Conference on Functional Programming (ICFP '10), ACM Press (2010) 63–74
8. Heeren, B.: Top Quality Type Error Messages. PhD thesis, Universiteit Utrecht, The Netherlands (2005) http://www.cs.uu.nl/people/bastiaan/phdthesis.
9. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In Horváth, Z., Zsók, V., Butterfield, A., eds.: Implementation of Functional Languages – IFL 2006. Volume 4449., Heidelberg, Springer Verlag (2007) 199 – 216
10. Hage, J., Heeren, B.: Strategies for solving constraints in type and effect systems. Electronic Notes in Theoretical Computer Science **236** (2009) 163 – 183 Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008).
11. el Boustani, N., Hage, J.: Improving type error messages for generic java. Higher-Order and Symbolic Computation **24**(1) (2012) 3–39 10.1007/s10990-011-9070-3.
12. Weijers, J., Hage, J., Holdermans, S.: Security type error diagnosis for higher-order, polymorphic languages. In: Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation. PEPM '13, New York, NY, USA, ACM (2013) 3–12

13. Hage, J., Harju, T.: A characterization of acyclic switching classes using forbidden subgraphs. SIAM Journal on Discrete Mathematics **18**(1) (2004) 159 – 176
14. Hage, J., Harju, T., Welzl, E.: Euler graphs, triangle-free graphs and bipartite graphs in switching classes. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Graph Transformation, First Int. Conf, ICGT 2002. Volume 2505 of Lecture Notes in Computer Science., Berlin, Springer Verlag (2002) 48–60
15. Hage, J., Harju, T.: The size of switching classes with skew gains. Discrete Math. **215** (2000) 81 – 92
16. Hage, J.: The membership problem for switching classes with skew gains. Fundamenta Informaticae **39**(4) (1999) 375–387
17. Hage, J., Harju, T.: Acyclicity of switching classes. European J. Combin. **19** (1998) 321–327