# Efficiently Executable Sets used by FireEye

FireEye Formal Methods Team Dresden*

26th May 2016

## 1 Introduction

FireEye provides cybersecurity solutions for detecting, preventing and resolving cyber-attacks. In order to increase the confidence in many of our products we perform proofs of security-critical software components in parallel with development of the components themselves. We carry out the proofs in a Coq [2] model that follows the structure of the implementation, but is much more abstract. The resulting gap is bridged by model-based testing.

Our testing framework is developed in OCaml and hinges on Coq's code extraction to obtain an executable version of the model. We execute millions of test cases, each with a state comprising thousands of objects. Thus, performance is a critical aspect: we need to fine-tune the used data structures based on the frequency of the operations performed.

Coq's sets library, while well-suited for mathematical reasoning, proved to be a bottleneck for our testing framework. In this note, we describe an extension of this library that drastically improves its usefulness in our development.

## 2 Fine Tuned Set Implementations for Code Extraction

Coq (as of version 8.4pl6) provides the module `MSetInterface` which contains interfaces for sets. There are implementations using sorted and unsorted lists (both without duplicates) as well as AVL and RBT trees for these interfaces. While these implementations—particularly the ones based on binary trees— extract to reasonably efficient code, we nevertheless were struggling with performance issues. Profiling revealed several problems related to the extracted set code: large sets of integers use a lot of space; slow insert and union operations are a huge issue for one of our algorithms and a slow specialised filter operation for another one. In the following we explain how we solved each of these challenges by providing general purpose extensions to Coq's sets library as well as special purpose set implementations.

**Interval Sets** In our setting, we need large sets of integers. They contain either only very few integers or nearly all the integers between 0 and 65535. The AVL tree set implementation was time efficient, but used large amounts of memory and, when marshalled, disc-space. By providing a set instantiation[1] that uses lists of intervals internally, we could shrink the memory and disc-space requirements drastically.

*`formal-methods@FireEye.com`, FireEye Technologie Deutschland GmbH, Wilsdruffer Strasse 27, 01067 Dresden, Germany, Amtsgericht Dresden HRB 33246, Geschäftsführer: Alexa King, Mark Hegarty

[1]module `MSetIntervals`

**Lists with Duplicates** Some of our algorithms collect a set of results by adding single results and combining result sets. They guarantee that results are not added multiple times to a set. For such requirements, lists are an efficient, commonly used datastructure. However, lists don't provide a high-level set view. On the other hand, all standard Coq set implementations are inefficient for our use case, because they perform an expensive membership test when inserting elements.

We provide an implementation of the set interfaces using lists with duplicates[2] and we allow fold to visit the same element multiple times. To achieve this, we removed the requirement that the element lists contain no duplicates from Coq's set interface `WSetsOn`[3].

**Fold With Abort** Folding over all elements of a set is a core operation, which has close ties to the concept of iterators in languages like C++. However, while iterators allow early abort, this is not possible with the standard fold operation. As a result, many efficient iterator algorithms become inefficient when implemented via folding.

Fold with abort operations (see e. g. [1]) are an answer to this issue. We provide interfaces for a family of such *fold with abort* operations and use these operations to define very efficient filter and existence check operations. The interfaces are instantiated for all of Coq's set implementations.

In our application, we use large sets of intervals implemented via AVL trees. Using fold with abort operations, we can very efficiently find all intervals overlapping with a given interval.

## 3 Conclusion

One of our typical test cases used to spend about 90 s in Coq extracted code (and 25 s in extra code). By using lists with duplicates, the runtime of the Coq extracted code could be reduced to about 9 s. Using interval sets reduced the overall disc-space required to store a test result to just about a quarter (while marginally improving the runtime). Finally, using fold with abort operations reduced the needed runtime to less than a second for the Coq extracted code. We believe that the set implementations used to achieve these improvements are useful in general and therefore provide them to the community [3].

The issues described here for the set interfaces are an instance of a more general issue we have encountered at FireEye. Many of Coq's libraries are aimed at theorem proving and provide good mathematical abstractions. However, they fall short of providing efficiently executable extracted code. We believe that this is an important deficiency of the Coq library and we would be delighted if the work presented here calls attention to this problem and becomes a first step towards providing a comprehensive library fine-tuned for code extraction.

## References

[1] Peter Lammich and Andreas Lochbihler. The Isabelle Collections Framework. In *ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 339–354, 2010.

[2] The Coq development team. *The Coq proof assistant reference manual.* INRIA, 2012. Version 8.4.

[3] FireEye Formal Methods Team. *Efficiently Executable Sets Extension: Source Code and Documentation.* FireEye Technologie Deutschland GmbH, May 2016. Available at `https://github.com/fireeye/MSetsExtra`.

[2]module `MSetListWithDups`

[3]see new interface `WSetsOnWithDups`