Aspects and Data Refinement
(Extended Abstract)

Pavel Avgustinov\textsuperscript{1}, Eric Bodden\textsuperscript{2}, Elhar Hajiyev\textsuperscript{1}, Oege de Moor\textsuperscript{1},
Neil Ongkingco\textsuperscript{1}, Damien Sereni\textsuperscript{1}, Ganesh Sittampalam\textsuperscript{1}, and Julian Tibble\textsuperscript{1}

\textsuperscript{1} Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
\textsuperscript{2} School of Computer Science, McGill University,
Montréal, Québec H3A 2A7, Canada

Abstract. We give an introduction to aspect-oriented programming
from the viewpoint of data refinement. Some data refinements are conve-
niently expressed via aspects. Unlike traditional programming language
features for data refinement, aspects conceptually transform run-time
events, not compile-time programs.

1 Introduction

Data refinement is a powerful tool in program construction: we start with an
existing module, adding some new variables related to the existing ones via a
coupling invariant, and possibly adding new operations as well. Next we refine
each of the existing operations so that the coupling invariant is maintained.
Finally, if any existing variables have become redundant, they are removed \cite{1}.

The idea is pervasive, and it is no surprise, therefore, that numerous re-
searchers have attempted to capture it in a set of programming language features.
An early example of this trend can be found in the work of Bob Paige, who ad-
vocated the use of a program transformation system to achieve the desired effect
\cite{2}. The idea was again raised by David Gries and Dennis Volpano in their design
of the transform in the Polya programming language \cite{3}. Very recently, Annie
Liu and her coworkers \cite{4} breathed new life into this line of work by updating it
to the context of object-oriented programming.

All these systems are very powerful, and they are complete in that all data
refinements can be expressed, at least in principle. In another community, a set
of programming language features has been proposed that is less powerful, but
still suitable for direct expression of simple data refinements. These features are
collectively known under the name of ‘aspects’ \cite{5}.

In this talk, we shall examine some examples of data refinement expressed as
aspects. Conceptually aspects transform run-time computations, unlike the above
systems, which are all based on the idea of compile-time transformation. For ef-
ciency, aspect compilers do as much transformation as possible at compile-time
\cite{6}, but that is an implementation technique, not the semantics. We argue that to
write reusable data refinements, which are independent of the syntactic details of
the program being refined, the run-time view offered by aspects is preferable.
2 Data Refinement

Consider an interface in Java for bags (multisets) of integers; an example of such an interface is shown in Figure 1. It includes an operation that returns an iterator over the elements of a bag; the order of such an iteration is not further specified.

```
interface Bag {
    void add(int i);
    void remove(int i);
    java.util.Iterator iterator();
}
```

Fig. 1. Bag interface in Java

Now suppose we wish to augment this interface, and all classes that implement it, with an operation that returns the average of the bag of integers. A naive implementation would be to re-calculate the average each time, but that requires time proportional to the size of the bag.

To achieve a constant-time implementation of average, we introduce two new variables via data refinement, namely sum and size. The coupling invariant is that sum holds the sum of the abstract bag, and size the number of elements.

```
public aspect Average {
    private int Bag.sum;
    private int Bag.size;

    public float Bag.average() {
        return (size == 0 ? ((float)sum) / ((float)size) : 0);
    }

    after(Bag b, int i) returning() :
        execution(void Bag.add(int)) &&
        this(b) &&
        args(i) {
            b.sum += i;
            b.size += 1;
        }

    after(Bag b, int i) returning() :
        execution(void Bag.remove(int)) &&
        this(b) &&
        args(i) {
            b.sum -= i;
            b.size -= 1;
        }
}
```

Fig. 2. Aspect for data refinement