Aspects and Data Refinement*

Pavel Avgustinov[1], Eric Bodden[2], Elnar Hajiyev[1], Oege de Moor[1],
Neil Ongkingco[1], Damien Sereni[1], Ganesh Sittampalam[1], Julian Tibble[1]

[1] Programming Tools Group, Oxford University, United Kingdom
[2] Sable Research Group, McGill University, Canada

Abstract. We give an introduction to aspect-oriented programming from the viewpoint of data refinement. Some data refinements are conveniently 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 [8].

The idea is pervasive, and it is no surprise, therefore, that numerous researchers 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 advocated the use of a program transformation system to achieve the desired effect [9]. The idea was again raised by David Gries and Dennis Volpano in their design of the *transform* in the Polya programming language [3]. Very recently, Annie Liu and her coworkers [6] 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' [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 efficiency, aspect compilers do as much transformation as possible at compile-time [7], 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.

* This work was supported, in part, by IBM, and by EPSRC in the United Kingdom, and NSERC in Canada.

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 contant-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. Once we have these two variables, it is easy to define an efficient implementation of the *average* function, as it just returns their quotient (provided the bag is not empty). Of course *sum* and *size* have to be kept up-to-date when *add* and *remove* are called: these operations must be data-refined accordingly.

Figure 2 shows how to code this data refinement in AspectJ, an aspect-oriented extension of the Java programming language [4]. First note how it introduces the two new variables into all implementations of the *Bag* interface, on Lines 2 and 3. Next we define the new *average* operation, on Lines 4 to 6. The remainder of the aspect is devoted to refining the *add* operation (Lines 7–14) and the *remove* operation (Lines 15–22). Let us examine the refinement of *add* in a little bit more detail. It says that whenever we have completed executing the body of *add*, on a bag *b*, with argument *i*, the sum should be increased by *i* and the size should be increased by 1.

Note that the aspect is generic, in that it applies the data refinement to any implementation of the *Bag* interface. Obviously this is a desirable property, as we can now reuse the same piece of code without having to replay the same data refinement each time a new implementation of bags is introduced.

3 Compile-time transformations

An obvious way to view aspects is as program transformations, which insert extra code into an existing program. Indeed, that has been the prevailing view in all previous works that sought to provide language support for data refinement.

The disadvantage of such a wholly syntactic approach is that it is very hard to write reusable data refinements, that are independent of the implementation details of the program being refined. To illustrate, consider changing the original *Bag* interface by adding a method *addAll*(*Bag c*); this new method adds all elements of another bag *c*

```
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.

to the given *Bag*. Formally, the call $b.addAll(c)$ implements the assignment (writing $+$ for bag union)

$$b := b + c$$

Now consider how the aspect should be modified, if at all, to take account of *addAll*. First observe that if we know that *addAll* is always implemented by iterating over *c*, calling the *b.add* method, no changes to our aspect are necessary. It is conceivable, however, that a more efficient implementation is used. For instance, when both the collection and the bag happen to be stored as sorted lists, a simple list merge would be cheaper than repeated element insertions.

It follows that for the aspect to remain reusable across all implementations of the *Bag* interface, we need to implement the data refinement of the new *addAll* method separately:

```
  after(Bag b, Bag c) returning() :
    execution(void addAll(Bag)) &&
    this(b) &&
    args(c)
{
    b.sum += c.sum;
    b.size += c.size;
}
```

It is not enough to add this piece of code to our aspect, however. If *addAll* is implemented via repeated calls to *add*, we would now add the sum of *c* twice to that of *b*. The data refinement of *add* itself therefore needs to be amended. Intuitively it is clear what amendment is required: when *add* is executed at the top-level, we use the refined code described earlier, and when part of other routines in *Bag* (such as *addAll*), the unrefined version of *add* is used. But this is a run-time distinction and not a compile-time one.

4 Run-time transformations

Motivated by this type of example, the designers of AspectJ advocate that aspects are viewed as run-time observers, which intercept events based on their run-time characteristics. In our running example, we only want to transform top-level method executions: in particular, the data refinement should apply to *add* when called on its own, but not when it is called from within another method of *Bag* like *addAll*. To achieve that objective in AspectJ, we can add the conjunct

!**cflowbelow**(**execution**($*$ Bag .$*$(..)))

to the pattern of Lines 8–10 in Figure 2. In words, it says the currently executing method invocation is not properly nested inside another method of *Bag*. Specifying the same behaviour as a compile-time transformation could be exceedingly painful. The *cflowbelow* primitive requires, in general, run-time observation of the state of the program, in particular the control stack. However, in practice this can often be statically determined by control-flow analysis [2] for efficiency.

The view of a data refinement in this setting is that an aspect checks the coupling invariant, and when the invariant may be violated, the aspect runs some extra code to restore the invariant. Much remains to be done to arrive at this point, however, and the challenges include:

Completeness What class of data refinements can be expressed via aspects? The example in this abstract only illustrates adding code before or after an operation on an abstract data type, and on its own it is clearly not enough to express all data refinements. What is a minimal set of aspect-oriented features needed to achieve completeness?

Diminution We have ignored the process of *diminution*, where auxiliary variables are removed from a data-refined program. While it is tempting to just rely on mechanical dead-code elimination in a compiler, it is unlikely that will always succeed. Aspects do offer a feature (so-called *around advice*) where operations can be replaced by others, in particular by *skip*.

Semantic patterns The patterns of interception should be less syntactic in nature, instead expressing properties like: ‘the state of this object may have changed’. Again this is important for aspects to be reusable.

We are investigating these and other challenges related to the design and implementation of aspect-oriented programming languages in the *abc* project [1]. We hope others will join us in exploring this new area, and in developing a rigorous basis for the use of aspects in program construction.

Acknowledgements

Richard Bird, Carroll Morgan, Jeff Sanders and Bernard Sufrin provided helpful feedback on a draft of this abstract.

References

1. abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.
2. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In Vivek Sarkar and Mary W. Hall, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 117–128. ACM Press, 2005.
3. David Gries and Dennis M. Volpano. The transform — a new language construct. *Structured Programming*, 11(1):1–10, 1990.
4. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.
5. Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
6. Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005)*, pages 473–486. ACM Press, 2005.
7. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.
8. Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. 2nd edition, Prentice Hall, 1994. See: `http://users.comlab.ox.ac.uk/carroll.morgan/PfS/`.
9. Robert Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, 1986.