Booster 2

Manual and Reference Guide

James Welch et al.

Department of Computer Science

University of Oxford

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Chapter 2

The Booster Language

Part II

Using The Booster Language

Part III

Transformations and Heuristics

Chapter 3

Transformations

3.1 Introduction

A 'staged compilation process', similar to that described on Page 8 of [1].

3.2 Structure

- Parse
  - Initialize lookup table
  - Populate lookup table
- Elaborate
  - Insert 'this' (requires that every term is annotated with the class it is contained in, and requires a function to lookup method and attribute names)
  - Generate inputs and outputs
    - Infer Types
    - Deduce Types
  - Populate lookup table
  - Inputs and outputs
  - Qualified invariants
  - Class-based invariants
  - Expanded workflows
  - (Expand Method References)
  - (Expand inheritance)
- Compile
  - 'Program'

    - Calculate postcondition
    - 'WP'
- Simplify
- Translate

3.2.1 Parse

In this initial stage of the process, the tree of abstract terms which comprises the abstract syntax is inserted into a lookup table, for later reference. At this point, only simple inferences are made; any complicated inferences are left until the 'elaborate' part of the process.

Before the lookup table can be created, it must first be initialised, which is useful in later steps: when the lookup-table is pretty-printed after a stage, perhaps for debugging purposes, there is no danger of an uninitialised part of the table.

3.2.2 Elaborate

3.2.3 Compile

3.2.4 Simplify

3.2.5 Translate

3.3 Implementing in Spoofax

3.3.1 Explaining the Booster transformations

Spoofax strategies typically take a 'graph re-writing' pattern: the tree is iteratively recursed, and when a term matching a particular pattern is found, an action is performed - typically a re-written version of the term is replaced in the tree.

However, this approach is not ideal for Booster transformations. Perhaps fundamentally, this is because the model is a graph, not really a tree. When iterating the tree, it is important to have a great-deal of contextual information present - for example, the list of attributes and their types for each class. This contextual information cannot easily be passed as a parameter during the tree exploration: it *could* be placed as annotations at suitable nodes in the tree. The main problem is that often the model may need updating somewhere other than the node currently visited.

These concerns may be illustrated by considering the function which deals with the expansion of the inheritance hierarchy. When a node is found which satisfies the property that

The node is an attribute 'a' in a class 'c1' The class 'c2' is a sub-class of the class 'c1' The class 'c2' does not currently contain the attribute 'a'

In this case it is necessary to have the relevant contextual information available during the examination of a node (which attributes are in which classes (other than the current), and which classes are sub-classes of others. Furthermore, it is necessary to be able to update this information in such a way that it can be used for the remainder of the tree-traversal, for otherwise further applications of this rule may be unapplicable, or, worse, repeated for the same node under the same pattern matching.

Under these constraints, two implementation choices make themselves clear. The first is that a mutable 'lookup-table' is required: a function which may be updated to reflect the latest understanding and is always available within any context.

The second is that the use of annotations in transformations such as this is not immediately useful. Annotations will need re-using and

3.4 A Lookup Table

There's a blurry line between a system and a booster specification. Since there is only one System in scope at any one time, then the lookup table acts as a lookup function for 'System'.

These are the fields that are originally stored in the lookup table.

```
"Name" -> name

"SetDef" -> (name, [elements])

"Class" -> (name, ([subclasses], [attributes], [methods],
                                      [constraints], [workflows]))

"Attribute" -> ((cname, aname), (decorations, type,
                                      (opposite), minmult, maxmult))

"Method" -> ((cname, mname), (constraint, guardedCommand, exts))
```

Where `exts` in `method` is used solely in the elaboration phase, to keep track of which constraints from subclasses have been conjoined into the method definition.

Part IV

Appendices

Appendix A

The Booster Syntax

Appendix B

Transformation Reference Guide

Bibliography

[1] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.