Science of Computer Programming 114 (2015) 20–32

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Oberon-0 in Kiama

Anthony M. Sloane *, Matthew Roberts

Department of Computing, Macquarie University, Sydney, Australia

ARTICLE INFO

Article history:
Received 2 May 2014
Received in revised form 9 October 2015
Accepted 12 October 2015
Available online 19 October 2015

Keywords:
Oberon-0
Attribute grammars
Term rewriting
Scala
Mixins

ABSTRACT

The Kiama language processing library is a collection of domain-specific languages for software language processing embedded in the Scala programming language. The standard Scala parsing library is augmented by Kiama's facilities for defining attribute grammars, strategy-based rewriting rules and combinator-based pretty-printing. We describe how we used Kiama to implement an Oberon-0 compiler as part of the 2011 LDTA Tool Challenge. In addition, we explain how Scala enabled a modular approach to the challenge. Traits were used to define components that addressed the processing tasks for each Oberon-0 sub-language. Combining the traits as mixins yielded the challenge artefacts. We conclude by reflecting on the strengths and weaknesses of Kiama that were revealed by the challenge and point to some future directions.

1. Introduction

Kiama is a collection of domain-specific languages (DSLs) for software language processing [1]. Each DSL is simple and has a small implementation as a library for the Scala general-purpose language. Program representations are Scala data structures and Kiama-based code can be compiled and executed by the standard Scala implementation [2] and edited by standard Scala development environments. DSL-based code is augmented by normal Scala code where necessary. The result is a very lightweight approach to language processing that integrates easily with other code and libraries.

This paper describes how we used our Kiama language processing library to implement an Oberon-0 [3] compiler as part of the 2011 LDTA Tool Challenge. It can serve as a general introduction to Kiama but omits features that are not directly relevant to the challenge tasks.

We distinguish between the following kinds of processing component which are used in the Oberon-0 implementation:

- *Syntax analysers* that read program text and build *abstract syntax trees* (Section 2). We use combinators from the standard Scala library to write syntax analysers [2, chapter 33].
- *Semantic analysers* that decorate trees with information and check the conformance of that information to language rules (Sections 3 and 4). The decorations take the form of *attributes* associated with tree nodes. Kiama's *attribute grammar DSL* provides a notation for expressing attribute equations [4].
- *Transformers* that restructure trees while staying within a single syntax (Section 5). We write transformers using Kiama's *strategic term rewriting DSL* which is based on the Stratego language [1,5,6].

* Corresponding author.
E-mail addresses: Anthony.Sloane@mq.edu.au (A.M. Sloane), Matthew.Roberts@mq.edu.au (M. Roberts).

http://dx.doi.org/10.1016/j.scico.2015.10.010
0167-6423/

- *Translators* that convert a tree conforming to one syntax into a tree conforming to another syntax (Section 6). We write translators using normal mutually-recursive Scala functions.
- *Pretty printers* that convert trees into text (Section 6). We write rules that govern pretty printing using a Kiama version of a Haskell-based *pretty-printing DSL* [7].

Section 7 describes the overall structure of the Kiama Oberon-0 implementation, including how the components relate to each other and how they are combined to make the challenge artefacts. Each component of the Oberon-0 implementation is a separate Scala trait. We use the mixin facilities of Scala to assemble components into language implementations [2, chapter 12]. These modularity features of Scala are orthogonal to the processing facilities provided by the Kiama library which do not need to address modularity at all.

Section 8 steps back from the details to reflect on the end product and what it has to say about the strengths and weaknesses of the Kiama approach.

In some cases the presented code has been slightly simplified from the actual code to make presentation simpler. For example, where the compiler uses lazy values to avoid issues with initialisation order, we use plain values here since initialisation is a minor issue that is irrelevant to the main topic of the paper.

The complete code of the Oberon-0 implementation can be found in the Kiama distribution. Documentation, source code and installation instructions for Kiama can be obtained from https://bitbucket.org/inkytonik/kiama. This paper is based on version 1.6 of Kiama which was the current version at the time of the challenge. Some time has passed since that version, so to make the paper of most use to current readers we briefly mention improvements in later versions of Kiama which are relevant to the challenge tasks.

2. Trees and syntax analysis

The abstract syntax trees built by Kiama syntax analysers are typically defined using a standard object-oriented approach. Each non-terminal of the grammar is an abstract class with concrete sub-classes for each variant of that non-terminal. The concrete classes in the tree definition are defined using Scala case classes [2, chapter 15]. Case classes are normal classes, but also share some properties with algebraic data types in functional languages. For example, instances can be created without the `new` keyword, their fields are immutable by default, field values can be extracted using pattern matching, and they implement field-based equality. We will see examples of using case classes when we discuss the tree processing phases in subsequent sections. For now, here is the fragment of the tree definition that defines assignment statements and some forms of expression.

```
abstract class Statement extends SourceTree
case class Assignment (desig : Expression, exp : Expression)
  extends Statement

abstract class Expression extends SourceTree
case class IntExp (v : Int) extends Expression
case class AddExp (left : Expression, right : Expression)
  extends Expression
```

`SourceTree` is the base class of all nodes in the Oberon-0 source tree. It's sole purpose is to enable Kiama's `Attributable` trait to be mixed in to each tree class. `Attributable` must be present to enable the generic access features of Kiama's attribution library which we will use in Section 3 and Section 4. The requirement to mix in `Attributable` has been removed in Kiama version 2 as described in our paper on smoothly combining attribution and term rewriting [8].

The versions of the Oberon-0 compiler that output C code first construct a tree to represent that code. We use the term *source tree* to refer to the tree of the Oberon-0 program and *target tree* to refer to the tree of the C program. The principles for defining and building target trees are the same as for the source tree, except that target trees are produced by translators instead of by syntax analysers (Section 6).

Syntax analysers are written using the standard Scala packrat parsing library combinators [2, chapter 33]. Complex parsers can be constructed from basic ones in a style that mimics context-free grammar productions. For example, the following parser for assignment statements shows a typical use of the main combinators.

```
val assignment = (lhs ~ (":=" ~> expression)) ^^ Assignment
```

`lhs` and `expression` are parsers defined elsewhere. A literal string is converted implicitly into a parser that just accepts that string. The tilde operator `~` sequences two parsers to make a parser that returns a pair of the values returned by its operand parsers. The `~>` operator also sequences, but throws away the value of its left operand. The `^^` operator transforms the result of a parser using an arbitrary function. In the example the `Assignment` tree node constructor is used to build a tree node from the children nodes that are built by the `lhs` and `expression` parsers. Kiama provides implicit conversions