Oberon-0 in Kiama

Anthony M. Sloane[a,*], Matthew Roberts[a]

[a]Department of Computing, Macquarie University, Sydney, Australia

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 sublanguage. 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.

Keywords: Oberon-0, attribute grammars, term rewriting, Kiama, Scala, traits, mixins

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.

*Corresponding author
Email addresses: `Anthony.Sloane@mq.edu.au` (Anthony M. Sloane),

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].

- *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 to enable a constructor to be used directly like this. In plain Scala parser combinator code it would be necessary to match on the result of the parse and pass those results explicitly to the constructor.

The actual formalism supported by Scala's parsing library is parsing expression grammars [9]. The main difference from context-free grammars is that the choice (alternation) operator is ordered which means that productions are unambiguous. As a result, it is necessary to order alternatives so that less-specific alternatives come later than more specific ones, otherwise the latter would never be chosen. For example, in the following simplified excerpt from the expression grammar the factor alternative must come after the operator alternatives because a factor will begin the input matched by the alternatives.

```
val term : PackratParser[Expression] =
  term ~ ("*" ~> factor) ^^ MulExp |
  term ~ ("DIV" ~> factor) ^^ DivExp |
  factor
```

The term parser is required by Scala to have an explicit type since it is recursive.

There is no separate scanning phase. Parsers for lexical symbols are constructed directly from regular expressions. For example, an identifier is parsed by first rejecting keywords and then accepting suitable strings of characters.

```
val ident = not (keyword) ~> "[a-zA-Z][a-zA-Z0-9]*".r
```

(The r method of a string converts it to a regular expression.) Rejection of keywords is performed using the not combinator which fails if its argument parser succeeds. The keyword parser is defined to accept a keyword only if it does not occur as the prefix of an identifier.

White space is skipped by the library parsers before literals and before input that matches regular expressions. The Scala library supports the definition of white space using a regular expression. Oberon-0 white space includes comments that can nest, so standard regular expressions are not sufficient. Rather than define an awkward non-standard regular expression for nested comments, we use a Kiama extension of the Scala library that allows white space to be defined by a parser. Therefore, we define Oberon-0 white space as follows.

Matthew.Roberts@mq.edu.au (Matthew Roberts)

```
  val whitespaceParser = rep (whiteSpace | comment)
  val comment = "(*" ~ rep (not ("*)") ~ (comment | any)) ~ "*)"
```

`whiteSpace` is the default white space regular expression. The `any` parser accepts any character. A parser built by `rep` matches zero or more repetitions of its argument parser.

3. Name analysis

After syntax analysis, the compiler performs semantic analysis on the source tree to check whether the program conforms to the language rules. Semantic analysis is implemented using Kiama's attribute grammar DSL [4]. Each piece of information needed for the analysis is represented by the value of an attribute that is associated with the most relevant node in the tree. For example, a type will be computed for each expression and a representation of that type will be associated with the node that represents the expression. Kiama attribute values are stored outside the tree so the definition of the tree does not have to be changed if attributes are added or removed.

The interface to an analysis component is a single `errors` attribute that collects messages that result from that analysis. If there are no errors, the compiler driver passes the source tree to the transformation and code generation phases. If there are errors, they are reported and the compiler exits. Source text positions for error messages are obtained from a map between source tree nodes and positions. This map is automatically created by a small Kiama-provided adaptation of the Scala parsing library.

Attribution of a tree requires a traversal to visit the relevant nodes in the correct order, equations that specify how to calculate attribute values, and storage for the calculated values. Kiama uses a demand-driven evaluation approach so that attribute values are only calculated if they are needed. Attribute equations define the data dependencies between attribute occurrences and hence implicitly define the traversal order. Calculated values are cached so that they do not need to be re-calculated if they are demanded again. Overall, this evaluation approach means that an analysis developer can focus on the data dependencies; tree traversal and attribute storage are taken care of by the library.

Name analysis in the Oberon-0 compiler is implemented using a set of attributes that compute properties of the declarations and uses of identifiers in the source program. An environment is computed for each tree node that contains exactly those bindings that are visible to that node according to the scoping rules of Oberon-0. The environment is threaded through the tree using a chain of attributes that is modelled after similar constructs in dedicated attribute grammar languages [10]. An attribute chain encapsulates a pattern of attribution that reaches each node from its parent, visits each of its children recursively from left to right, then leaves the node to the parent. At each point along the way the value of the chain can just be passed along or a new value can be computed, usually based on the value of the chain coming into that point.

Kiama provides support for defining chains where the default threading behaviour is provided but can be overridden by defining partial functions that match on tree nodes. For example, the environment attribute `env` is defined as follows:

```
val env : Chain[SourceTree,Environment] =
  chain (envin, envout)
```

The functions `envin` and `envout` define custom behaviour on entry to and exit from the sub-tree at a node, respectively. These functions are partial; if they are not defined at a particular node then the value of the chain is passed on unchanged.

`envin` and `envout` cooperate to adjust the environment so that at any given node it is an accurate representation of the bindings that are in scope at that node. For example, the environment chain is defined by `envin` to start at the root of the tree (a module declaration) with a default environment `defenv` and to enter a new nested scope for the module. That processing is implemented by the first case of `envin`.

```
def envin (in : SourceTree => Environment) = {
  case _ : ModuleDecl => enter (defenv)
  case b : Block      => enter (in (b))
  ...
}
```

`envin` is defined using Scala's partial function literal syntax which uses pattern matching cases within a brace-delimited block. The second case of `envin` ensures that at each block node we enter a nested scope for that block. The outer block is obtained by applying the `in` function which gives access to the default value of the chain. Analogously, `envout` leaves nested environments as the chain passes module declaration and block nodes on the way out of a sub-tree.

```
def envout (out : SourceTree => Environment) = {
  case m : ModuleDecl => leave (out (m))
  case b : Block      => leave (out (b))
  case n @ IdnDef (i) => define (out (n), i, n->entity)
  ...
}
```

The third case of `envout` carries out the heart of the environment calculation at identifier definition nodes by adding the identifier and its entity (described below) to the topmost scope of the current environment. Operations on environments are provided by a Kiama module, including `enter`, `leave` and `define` used here, plus `isDefinedInScope` and `lookup` which are used below in the definition of `entity`.

At any node in the tree the environment chain can be accessed to obtain the visible bindings for that location in the program. In particular, we use the chain to compute the program entity that is represented by each identifier occurrence.

Entities can contain properties directly or have fields that refer to tree nodes from which properties can be derived. For example, a constant entity holds a reference to the constant declaration node from which the expression defining the constant can be obtained. The following code defines the `entity` attribute.

```
val entity : Identifier => Entity =
  attr {
    case n @ IdnDef (i) =>
      if (isDefinedInScope (n->(env.in), i))
        MultipleEntity ()
      else
        entityFromDecl (n, i)
    case n @ IdnUse (i) =>
      lookup (n->(env.in), i, UnknownEntity ())
  }
```

This attribute is typical of equations that match on the node at which the attribute is being calculated. The Kiama `attr` function takes a single argument that is a partial function defined by cases and wraps it with the demand-driven evaluation mechanism of attributes. The "brace and case" syntax of Scala's partial function literals makes an `attr` application look like a built-in construct when it is just a call to a Kiama method. Attribute occurrences are accessed in the definition of `entity` using the `->` infix operator, but they can also be accessed in a functional style. In these equations, `env.in` is an attribute that is the value of the environment chain as it enters a node. In each case, the name `n` is bound to the matched node via the `n @` notation preceding the pattern. The expression `n->(env.in)` obtains the value of that attribute at node `n`, as would `env.in(n)`.

In the `entity` attribute definition we associate a representation of a program entity with each identifier occurrence. At every node that represents the declaration of an identifier (`IdnDef`), we check using `isDefinedInScope` to see if the identifier is already present in the topmost scope of the environment. If this is the case, the identifier has been defined more than once in that scope so we represent it by a special `MultipleEntity`. If the identifier is new in the current scope, we calculate an entity for it using the `entityFromDecl` method. Back in `entity`, at every node that represents a use of an identifier (`IdnUse`), the environment is searched using `lookup` to try to find a binding for the identifier. If a binding cannot be found we represent its entity by a special `UnknownEntity`.

The `entity` attribute is defined by two cases: one for defining occurrences of identifiers (`IdnDef` nodes) and one for uses (`IdnUse` nodes). We are relying here on defining occurrences and uses being syntactically distinguished. An alternative is to use a single node type for all identifier occurrences and use some other mechanism to examine the context of those occurrences to tell defining occurrences from uses. This alternative approach would certainly be possible using Kiama but we used the syntactic approach since it is possible for Oberon-0 and keeps the definition of `entity` simple.

Entities form the basis of name analysis checking. Most semantic checks are simple conditions on attributes. For example, name analysis for the L1 language checks that no identifier is multiply-defined and that each use has an associated declaration. These checks are implemented by the following code in the `errorsDef` method. `n` is the node that is being checked.

```
n match {
  case d @ IdnDef (i) if d->entity == MultipleEntity () =>
    message (d, s"$i is already declared")
  case u @ IdnUse (i) if u->entity == UnknownEntity () =>
    message (u, s"$i is not declared")
  ...
}
```

The `message` method returns a message at the source text location of the tree node that is passed as its first argument.

Finally, all of the errors for the tree are collected to compute the `errors` attribute for a tree. The implementation of `errors` uses Kiama's strategic programming facility to generically traverse the tree so that all nodes will be checked.

```
val errors =
  attr (collectall {
    case n : SourceTree =>
      errorsDef (n)
  })
```

`collectall` visits every node in the tree and accumulates the values produced by its function argument. In this case we just call `errorsDef` at each node and accumulate the messages that it returns. The `SourceTree` type annotation is necessary to restrict the type for the `errorsDef` call since `collectall` assumes all nodes are of the Scala root type `Any`.

This error collection scheme is designed to be extensible. We do not anticipate the conditions that might be checked or the errors that might be generated, since `errorsDef` is free to do what it likes. Since the check for errors reaches every node, there is no limit on future extensions. An extension can override `errorsDef` to provide new cases and call the overridden version to retain errors from the extended language. More discussion on our approach to extensibility can be found in Section 7.

Name analysis at one level extends easily to other language levels. If no new declaration or scoping constructs are added, no changes need to be made. The environment chain will simply traverse into the new constructs and the existing checks will be performed. New declarations and scoping constructs can be accommodated by overriding the definitions of the `entityFromDecl` method and the environment chain. The overridden method is called to access processing for old constructs. For example, in the L3 name analyser the chain is adjusted to take account of procedure scopes. New kinds of entities are added to represent procedures and parameters. All environment processing, the entities and the checks from earlier levels are reused by the L3 analyser without change. (See Section 7 for more details on how the components are combined in a modular way.)

4. Type checking

The type analysis components of the Oberon-0 compiler are defined in a similar way to the name analysis components. Each expression has a `tipe`[1] attribute that calculates its type from its contents, and an `exptype` attribute that calculates the type expected by the context.

For example, removing irrelevant details, the `tipe` attribute in the L1 type analyser matches the following cases to an expression node `n` to assign types to relational, logical, arithmetic and identifier use expressions:

```
n match {
  case _ : EqExp | ... | _ : OrExp | ... =>
    booleanType
  case _ : AddExp | _ : MulExp | ... =>
    integerType
  case IdnExp (u) =>
    u->idntype
  ...
}
```

The first two cases perform type tests on the tree nodes and choose a Boolean or integer type as appropriate. The third cases uses the `idntype` attribute which accesses the entity of an identifier use (as determined by name analysis) to determine its type from its declaration.

The expected type of an expression is calculated by pattern matching on the expression's parent node since it defines the context. Each node is equipped with a parent reference by virtue of its class inheriting from `Attributable` as discussed in Section 2. The parent references are set by a pass over the tree before attribution begins. (As noted earlier, Kiama 2 removes `Attributable` and provides a safer mechanism for matching on the relationships between nodes [8].) For example, if the parent of an expression node `n` is an OR, AND or NOT node then the expected type of `n` is Boolean. The expected type of the right-hand side of an assignment is the type of the left-hand side. These computations are implemented by the following cases in the definition of the `exptype` attribute:

```
n.parent match {
  case _ : OrExp | _ : AndExp | _ : NotExp =>
    booleanType
  case Assignment (d, _) =>
```

[1]`type` is a reserved word in Scala.

```
      d->tipe
    ...
  }
```

The parent reference provides a convenient way to access the context of a node and to define contextual attributes such as the expected type. Since our tree structures are defined by collections of Scala case classes there is no static way to ensure that the parent of a node is of a particular type, as would be possible with a proper grammar analysis. For this reason the type of the parent reference is just the base type of all tree nodes. Hence, the use of parent opens an aspect of dynamic typing in the compiler implementation and we rely on testing to confirm appropriate behaviour.

The L1 type analyser checks one simple condition: that the type of an expression is compatible with its expected type. The exact definition of compatibility is provided by the `isCompatible` method. The definition of this method is overridden in other language levels as the notion of compatibility changes. The check itself is left unchanged.

The type analyser cooperates with the name analyser to avoid reporting spurious errors. For example, a name that is assigned an `UnknownEntity` results in a name analysis error. It is useless to also report type analysis errors in expressions that use such a name. The type analyser assigns an "unknown" type to identifier use expressions that have unknown entities. The `isCompatible` method regards unknown types to be compatible with any other type.

5. Source-to-source transformation

In the Oberon-0 compiler the following source-to-source transformations are performed:

1. Make user-level names unique to assist other transformations such as lifting. This transformation is placed at L1 since it only depends on the constructs that represent defining and applied occurrences of identifiers.
2. Replace L2 `FOR` and `CASE` statements with their L1 equivalents as required by T4.
3. Lift nested declarations to the top-level as required by T4 and T5. Although the challenge required this only for L3 to handle nested procedures, we place it at L2 since it works for any declaration, not just declarations of procedures. It is also used in our compiler to lift declarations out of nested blocks that are created by the `FOR` and `CASE` transformation. We discuss the further in the following.

We now discuss the desugaring of L2 `FOR` and `CASE` statements in more detail to illustrate how a source-to-source transformation is achieved. Semantic analysis of these constructs is performed as described in the previous two sections. Between analysis and code generation, we *desugar* these constructs into equivalent simpler constructs. `FOR` statements are transformed into `WHILE` statements. `CASE` statements are transformed into cascades of `IF` statements.

In this compiler, a *transformation* component implements a single method that takes the root of a source tree and returns a possibly transformed source tree. As for the analysis phases, the transformer for one language level calls the transformer for the next lower level, so the transformations are composed without requiring any knowledge of each other. Transformations stay within the source language, in contrast to *translation* components that turn a source tree into a target tree, thereby changing languages (Section 6).

Transformations are implemented using Kiama's strategy-based term rewriting library. Here we are using strategies in a type-preserving fashion (i.e., within a single syntax) and Kiama uses Scala's static type system to ensure that rewrites do actually preserve types (in contrast to the more dynamically-typed Stratego, for example). It is also possible to use Kiama's strategies in a statically-checked type-unifying fashion. For example, the `collectall` combinator used to collect errors in Section 3 operates on the Oberon-0 syntax but produces collections of errors.

Another form of strategy-based term rewriting allows terms to contain constructs from both source and target languages while a translation is in progress. Since Scala is statically typed we can only support this form of rewriting if the source and target types are combined into one which is less than ideal. For this reason, we usually write translations as normal recursive functions instead of using rewriting. See Section 6 for details of how this is done in the Oberon-0 code generator.

The L2 transformation of `FOR` and `CASE` statements is defined simply as

```
everywhere (desugarFor + desugarCase)
```

which at every node in the tree first tries to apply the transformation to desugar `FOR` statements and, if that fails, tries to apply the transformation to desugar `CASE` statements. (An optimisation could easily be applied to attempt this transformation only in sub-trees where statements can occur.)

The actual `FOR` transformation is defined by the `desugarFor` rewrite rule which simply pattern matches and returns the replacement tree fragment (Figure 1). A `FOR` statement is translated into a block containing a declaration of a variable to hold the limit of the iteration and a `WHILE` statement to implement the loop. This approach means that we do not need a different variable name for each `FOR` loop, since the scoping rules ensure that each loop will refer to the correct variable even if `FOR` loops are nested. Note that the Oberon-0 source language does not allow nested blocks, but our abstract syntax does so to make these kinds of transformations easier to define.

A complication in this rule when using Kiama 1.6 is that we must be careful not to insert the same node in more than one place. For example, when we want to refer to the loop control variable in the assignment statement via `idnexp`, we must use a different node to the appearance of that variable in the condition. Cloning is necessary because attributes are cached using the node identity as the key and the two nodes will usually have some different attribute values. In fact, the compiler is conservative when it comes to reuse of attribute values.

```
val desugarFor =
  rule {
    case ForStatement (idnexp, lower, upper, optby,
                       Block (Nil, stmts)) =>
      val limvarname = "limit"
      val limexp = IdnExp (IdnUse (limvarname))
      val incval = optby.map (_->value).getOrElse (1)
      val rincval = IntExp (incval)
      val cond = if (incval >= 0)
                     LeExp (idnexp, limexp)
                 else
                     GeExp (idnexp, limexp)
      Block (
        List (
          VarDecl (List (IdnDef (limvarname)),
                   NamedType (IdnUse ("INTEGER")))
        ),
        List (
          Assignment (clone (idnexp), lower),
          Assignment (clone (limexp), upper),
          WhileStatement (cond,
            Block (
              Nil,
              stmts :+
                Assignment (clone (idnexp),
                            AddExp (clone (idnexp), rincval))))
        )
      )
  }
```

Figure 1: Kiama desugaring of a `FOR` statement into a block containing a `WHILE` statement. The underscore in the definition of `incval` turns the mapped expression into a function of one argument that places that argument at the position of the underscore. The binary operator `:+` defined on collections appends an item to the collection.

The attribute caches are cleared after any transformation to ensure that old, potentially incorrect values are not carried over to the new tree.

These restrictions imposed by Kiama 1.6 have been removed in Kiama version 2. Specifically, our work on combining attribution and rewriting showed how to separate tree identity from the underlying node structure to remove confusion about which structure is being used to compute which attribute occurrences [8]. This change removes the need to explicitly clear attribute caches. Kiama 2 also lazily clones trees where necessary so that explicit cloning is now not necessary in rewrite rules such as that in Figure 1.

6. Code generation

The code generation components of the compiler comprise three main parts: adjustments to the source tree to prepare for translation, translation from the source tree to the target tree, and pretty-printing of the target tree. There is one code generation component for each language level. The code generator for one level invokes the code generator for the next lower level to deal with constructs at that lower level.

At some language levels the source tree is more permissive than the target tree and must be simplified before a straight-forward translation can be performed. For example, the source tree for L2 may contain nested blocks that arise from the desugaring of `FOR` and `CASE` statements. Similarly, L3 contains procedure declarations that can appear at any nesting level. The target tree requires C function declarations to appear at the top level. As noted in Section 5 source-to-source transformations are used to make names unique and to perform a simple version of lambda lifting that moves inner declarations to the top level of the source program.

Once the source tree has been adjusted in this way, it can be easily translated into a target tree. Oberon-0 constructs translate directly across to equivalent C ones. The translation is performed by a collection of mutually recursive functions that deal with each source tree construct and produce the corresponding target tree construct. For example, Figure 2 shows the method that translates L1 declarations to C declarations. The method returns a sequence since a single Oberon-0 variable declaration could declare more than one variable and we translate them to separate C variable declarations. User-level identifiers are mangled to avoid clashes with pre-defined C names. The translator uses the attributes `value`, to obtain the value of a constant initialising expression, and `deftype`, to obtain a defined source type that must be translated to a target type.

Finally, the target tree is printed to produce the C program text. Kiama has a pretty-printing domain-specific language based on a Haskell library [7]. Figure 3 shows part of a method that translates C target trees into pretty-printing documents. Combinators allow us to specify concrete translations such as inserting a semi-colon or comma, as well as patterns of printing such as wrapping in delimiters (`parens`, `braces`), layout using separators (`hsep`, `vsep`), and nesting to increase indentation (`nest`). Constraints such as possible positions for line

```
def translate (d : Declaration) : Seq[CDeclaration] =
  d match {
    case ConstDecl (IdnDef (i), e) =>
      Seq (CInitDecl (CVarDecl (mangle (i), CIntType ()),
                      CIntExp (e->value)))
    case TypeDecl (IdnDef (i), t) =>
      Seq (CTypeDef (CVarDecl (mangle (i),
                     translate (t->deftype))))
    case VarDecl (is, td) =>
      val t = td->deftype
      is map {
        case IdnDef (i) =>
          CVarDecl (mangle (i), translate (t))
      }
  }
```

Figure 2: Translation of L1 declarations.

breaks are also indicated using combinators (`line`). Once a complete pretty-printing document has been obtained, it is linearised subject to the specified constraints to obtain the target text.

7. Artefacts

The Kiama Oberon-0 challenge artefacts are constructed by combining components that each address a single processing task for a particular language. The components, their compositions into tasks and artefacts, and the sizes of all of these pieces are summarised in Figure 4. Each non-empty cell in the top table represents a task and language-specific component, implemented by a separate Scala trait. The bottom table aggregates the component sizes according to the artefacts in which they belong and includes support code. For example, artefact A1 includes all components from the L1 and L2 levels for tasks T1 and T2, plus driver code. (Our implementation actually has two simpler language levels, Base and L0, below the first challenge level that are used to separate out basic constructs. In this paper we aggregate Base and L0 into L1 so as to keep to the challenge levels.)

In addition to the components shown in the top table of Figure 4, each artefact includes two drivers. A generic compiler driver handles common tasks such as command-line option processing, file handling and printing output as required by the challenge. Each artefact also has a custom driver object that composes a compiler driver with the relevant main components. The bottom table in Figure 4 shows the sizes of each of these pieces. Note that this table is double-counting compiler driver code that is reused in different artefacts.

To illustrate how an artefact is composed from processing components, consider the driver for the A4 artefact (Figure 5). The `A4` object is a singleton that

```
def toDoc (n : CTree) : Doc =
  n match {

    case CProgram (is, ds) =>
      vsep (is map toDoc) <@>
      vsep (ds map toDoc, semi)

    case CInclude (s) =>
      s"#include $s"

    case CFunctionDecl (d, args, b) =>
      toDoc (d) <+>
      parens (hsep (args map toDoc, comma)) <+>
      toDoc (b)

    case CBlock (ds, ss) =>
      braces (nest (lterm (ds map toDoc, semi) <>
                    lsep (ss map toDoc, empty)) <>
              line)
    ...
  }
```

Figure 3: Pretty-printing of C constructs.

is the main program for this artefact. It is defined in terms of the `A4Phases` trait that represents the composed components. `artefact`, `langlevel` and `tasklevel` are identification fields that are used by our testing framework. The phases trait is not strictly necessary to define the artefacts; we could just make the A4 object have the definition shown for the `A4Phases` trait. Having a separate trait is useful for testing, however, since the testing class can mix in the trait and extend with tests.

Figure 6 shows how the components of the A4 artefact relate to components at the different language levels. The layout mimics that of Figure 4: languages on the x axis and components on the y axis. The square components in Figure 6 correspond to the traits that are mixed together to make the A4 artefact in Figure 5, omitting the driver component. The A4 components use many other components via extension (inheritance) or by mixing them in. Trait extension and mixin composition together define the complete functionality of the artefact.

Scala uses a linearisation algorithm to transform the acyclic extension and mix-in relationships into a linear sequence of traits that form the final composition of an artefact [11]. Linearisation enables us to unambiguously talk about the superclass of a trait in a composition even though the trait itself does not declare that it extends that superclass. This ability to independently define traits and compose them in many different ways is the key to Scala's modularity approach. The details of the linearisation algorithm are not important here.

Task	*Component*	*Language*				*Total*
		L1	*L2*	*L3*	*L4*	
T1	Oberon-0 tree	84	15	14	12	*125*
	Syntax analyser	152	35	32	31	*250*
	Oberon-0 printer	122	41	25	51	*239*
	Sub-total	*358*	*91*	*71*	*94*	*614*
T2	Name analyser	205	17	81	22	*325*
T3	Type analyser	94	23	84	108	*309*
T4	Lifter		23			*23*
	Desugarer	40	83			*123*
	Sub-total	*40*	*106*			*146*
T5	C tree	93		18	8	*119*
	Code generator	133		84	35	*252*
	C printer	103		25	19	*147*
	Sub-total	*329*		*127*	*62*	*518*
	Total	*1026*	*237*	*363*	*286*	*1912*

	Artefact				
	A1	*A2a*	*A2b*	*A3*	*A4*
Components	671	823	788	1024	1912
Compiler driver	90	90	90	113	133
Artefact driver	12	12	13	15	17
Total	773	925	891	1152	2062

Figure 4: Sizes of Kiama Oberon-0 compiler components grouped by challenge task and language level (top) and artefacts, including drivers (bottom). Size is measured in non-blank, non-commented lines of Scala code.

```
trait A4Phases extends base.TranslatingDriver
    with L4.SyntaxAnalyser
    with L4.source.PrettyPrinter
    with L4.NameAnalyser
    with L4.TypeAnalyser
    with L2.Lifter
    with L2.Desugarer
    with L4.CCodeGenerator
    with L4.c.PrettyPrinter {

    def artefact = "A4"
    def langlevel = 4
    def tasklevel = 6

}

object A4 extends A4Phases
```

Figure 5: The artefact driver for A4 which composes a driver appropriate for artefacts that translate, the L2 lifter and desugarer components, and the L4 components.

It suffices to know that if T extends or mixes-in U then T will precede U in any linearisation where they both occur. Also, a trait will only appear once in the linearisation even if it is extended or mixed-in by more than one other trait.

The linearisation defines the behaviour when a particular component calls `super.m` for some processing method `m`. For example, recall the `errorsDef` method described in Section 3. Each name and type analyser defines this method to contribute its errors. An implementation of `errorsDef` does its processing and calls `super.errorsDef` to handle other errors. For example, in the A4 artefact the L4 type analyser extends the L3 type analyser so we know that calling `super.errorsDef` in the former will eventually call `errorsDef` in the latter, possible with intervening error processing from other components. This approach means that a particular component doesn't need to know which other error-providing components are used, unless it directly extends them or mixes them in. A component can be composed with many such components to form different artefacts without changing the code of the components.

A similar method-overriding approach is used to compose source-to-source transformation and code generation components. For example, the interface to a source-to-source transformation component is a `transform` method that takes a source tree and returns a (possibly transformed) source tree. Transformations are extended by overriding `transform` to add more processing and by calling `super.transform` to invoke lower-level transformations. Exactly which superclass's method is called depends on a particular linearisation. For example, in the A4 artefact when the L2 lifter calls `super.transform` it will call the `transform` method in the L2 desugarer, since the L2 desugarer is the next

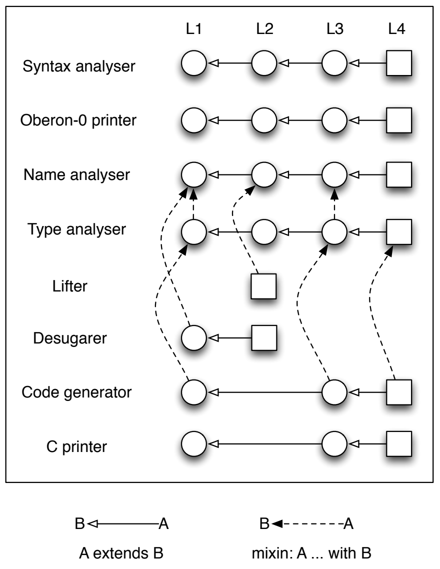

Figure 6: Component relationships in the A4 artefact. The A4 components are squares. Other components that are extended or mixed-in are circles. Solid lines indicate where a component extends another; dashed lines indicate a mix-in relationship.

transformation component in the A4 artefact linearisation. When the L2 desugarer calls `super.transform` it will call the method in the L1 desugarer, again courtesy of the linearisation.

This composition approach is based on extending methods. A variant is required to extend values which are used in the compiler to define parsers (Section 2) and attributes for semantic analysis (Section 3 and Section 4). It is tempting to try an approach to composition that just uses `super` references to access values in other components, as was done for methods. However, this approach doesn't work since Scala doesn't allow `super` to be used on values. Instead, we only use a value at the lowest level and dispatch to associated methods that can then be extended as described above.

For example, we define a single L1-level `statement` value to be the parser for the simplest kinds of statements. This value simply calls an associated method `statementDef` method which actually defines the parser.

```
val statement = statementDef
def statementDef = ...
```

To extend the parsing of statements we simply override `statementDef` as in the error handling, transformation and translation cases. Note that an overriding component doesn't have to just add alternatives to a parser, it can replace the parser entirely. We use this approach when adding procedure declarations to replace the previous syntax which only included constant, type and variable declarations. Adding a new alternative is not necessary since we can use the overridden definition as part of a new one that adds procedures, as in:

```
override def declarationsDef =
  super.declarationsDef ~ rep (procedureDeclaration <~ ";")
```

(where we omit actions). It would be possible to get the same effect by adding an alternative as long as the new alternative was added before the overridden ones to avoid ambiguity. Clearly this approach has limitations since in an embedding approach it is non-trivial to insert new alternatives in the middle of overridden parser definitions or adjust overridden definitions without breaking the encapsulation of the parser implementation.

The approach of calling out to a method is also used to extend attribute values. The method associated with an attribute defines the equations by cases for a particular component and extensions define cases for new equations. An important consequence of this approach is that we get only one cache per attribute (since there is only a value at the L1 level) instead of one per extension.

While the composition that results in the final A4 artefact is complex when regarded as a whole, each aspect of this composition is simple and allows us to separate the code for each component. Each trait extends or mixes-in just the traits that it needs to do its work. For example, the L3 syntax analysis component just extends the L2 syntax analysis component by augmenting existing parsers or adding new ones for L3 constructs. The L3 type analyser extends the L2 type analyser and mixes in the L3 name analyser since it needs to use entity information for the L3 level to determine L3 types. When an artefact is assembled out of traits, we just choose the functionality that we need in the artefact and mix them together. Scala's linearisation algorithm determines an order that is compatible with all of these local dependencies.

Overall, this design leads to an extremely flexible structure in which each component is focused on a single task and sub-language. The decision about how to compose the pieces is left until the traits are mixed together. The Scala compiler statically ensures that a particular composition is legal.

8. Reflection

A core tenet of the Kiama design is to reuse as much as possible from the host Scala language. Aspects such as data structure implementation, modularity and composition are left to Scala. Kiama itself is thereby free to focus on core language processing issues. The Oberon-0 compiler illustrates this reliance on Scala quite well. It makes heavy of use of powerful features including case classes and pattern matching, as well as more mundane ones such as expression syntax and definition mechanisms such as values and methods. The compiler would be much harder to write in a language that didn't provide these features. Sometimes standard programming approaches that would be provided by any language were sufficient. For example, just using recursive translation methods in the code generator is appropriate since they provide just the pattern of computation that is needed.

The trait and mixin features provided by Scala proved to be invaluable for structuring the compiler code. We were able to keep components separate from each other and assemble them into artefacts in flexible ways. This was our first non-trivial use of Scala mixins for this purpose and we believe they passed with flying colours.

This experience reinforced our belief that there is no need for domain-specific languages to explicitly address "bigger" issues such as modularity and composition. By embedding DSLs in a sufficiently powerful host language, we can keep the DSLs focussed on the problem areas that they are aiming to address. This situation contrasts with that of external DSLs where there is always a tendency to extend the languages beyond their problem domains by adding general-purpose features. For example, external DSLs frequently invent expression languages and features for modularity and namespace control, none of which are central to the purpose of the DSL. The DSL implementation is more complicated than it has to be and users have to learn features that are similar to other languages but are often idiosyncratic.

Of course, embedding DSLs as libraries in a host language implies that they must live within the limitations of the host. For example, the domain-specific analysis and optimisation that we can do for Kiama DSLs is limited, since all of the processing is performed by the Scala compiler. The result is that some checks that would ideally be performed statically are deferred until run-time. For example, it is not possible to statically analyse whether a Kiama attribute occurrence depends on itself, a check that many attribute grammar systems make. Further, it is not possible to implicitly compose definitions by reusing names as is often done in external domain-specific languages. Kiama requires explicit composition since Scala does.

These restrictions did not impede the development of the Oberon-0 compiler, and might even be considered an advantage since features like implicit composition can make code harder to understand. Notwithstanding the need to live within Scala's limitations, the notations that the Scala parsing library and Kiama provide are close enough to domain-specific ideals to yield clear code. We have not observed any performance issues in this compiler arising from the lack of domain-specific optimisations.

Recent versions of Scala add powerful macro processing facilities [12]. At present, we do not perform any meta-level manipulation of Kiama-based programs using macros, except for extraction of user-level names for use in debugging and profiling [13]. Therefore, we retain a relatively simple, lightweight implementation. Relying on macros for domain-specific processing would tie Kiama closely to the internals of the Scala implementation, which we prefer to avoid.

The exercise of building this compiler made it clear that Kiama could be more helpful in a number of areas. Combining attribution and rewriting can be problematic. For example, if you have already calculated an attribute of a tree node and that node is shared by a rewritten tree, what happens to the attributes? When using Kiama 1.6 our solution is to clearly separate attribution and rewriting tasks, resetting all attribute values between major processing tasks. In work conducted since the challenge we have developed techniques for systematically separating the relationships between tree nodes and their identity-based use in particular structures [8]. Kiama 2 incorporates this approach and therefore makes it easier to safely combine attribution and rewriting.

Tension between attribution and rewriting is also apparent when a node is shared in the output of a rewrite rule. Since the attribution implementation is based on the identity of a node, in Kiama 1.6 it is necessary to manually clone nodes rather than use them in more than one place. Kiama 2 lazily clones nodes so accidentally forgetting to clone is not possible.

Kiama does not currently provide concrete object syntax for use in pattern matching and tree construction. Adding it would make code such as that in Figure 1 much easier to write and understand. Scala's recent addition of powerful string interpolation features offers a direction for support of concrete object syntax.

9. Conclusions

Overall, we found the challenge to be a very useful exercise. Compiling Oberon-0 is not an extremely difficult task, but it incorporates aspects of any compilation process except for optimisation. Building the compiler forced us to explore composition of components in a more complex setting than we had tried before. We believe that the result shows the efficacy of the Kiama approach that mixes general purpose programming with library capabilities for common language processing tasks.

Acknowledgements

Thanks to Niklas Fors, Görel Hedin, Tijs van der Storm and the anonymous reviewers for comments that considerably improved the paper.

References

[1] A. M. Sloane, Lightweight Language Processing in Kiama, in: Generative and Transformational Techniques in Software Engineering III, vol. 6491 of *Lecture Notes in Computer Science*, Springer, 408–425, 2011.

[2] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima Press, 2 edn., 2010.

[3] N. Wirth, Compiler Construction, Addison-Wesley, 1996.

[4] A. M. Sloane, L. C. L. Kats, E. Visser, A pure embedding of attribute grammars, Science of Computer Programming 78 (2013) 1752–1769.

[5] E. Visser, Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9, in: Domain-Specific Program Generation, vol. 3016 of *Lecture Notes in Computer Science*, Springer, 216–238, 2004.

[6] E. Visser, WebDSL: A Case Study in Domain-Specific Language Engineering, in: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, vol. 5235 of *Lecture Notes in Computer Science*, Springer, 291–373, 2007.

[7] S. Swierstra, O. Chitil, Linear, bounded, functional pretty-printing, Journal of Functional Programming 19 (1) (2008) 1–16.

[8] A. M. Sloane, M. Roberts, L. G. C. Hamey, Respect Your Parents: How Attribution and Rewriting Can Get Along, in: Proceedings of the International Conference on Software Language Engineering, vol. 8706 of *Lecture Notes in Computer Science*, 191–210, 2014.

[9] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 39, ACM Press New York, NY, USA, 111–122, 2004.

[10] U. Kastens, W. M. Waite, Modularity and Reusability in Attribute Grammars, Acta Informatica 31 (1994) 601–627.

[11] M. Odersky, M. Zenger, Scalable Component Abstractions, Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (2005) 41–57.

[12] E. Burmako, Scala Macros: Let Our Powers Combine!, in: Proceedings of the 4th Annual Scala Workshop, ACM, 2013.

[13] A. M. Sloane, M. Roberts, Domain-specific program profiling and its application to attribute grammars and term rewriting, Science of Computer Programming (2014) 488–510.