

Compositional compiler construction: Oberon0



Marcos Viera^a, S. Doaitse Swierstra^b

^a Instituto de Computación, Universidad de la República, Montevideo, Uruguay

^b Department of Computer Science, Utrecht University, Utrecht, The Netherlands

ARTICLE INFO

Article history:

Received 30 April 2014

Received in revised form 10 March 2015

Accepted 12 October 2015

Available online 23 October 2015

Keywords:

First-class attribute grammars

First-class grammars

Haskell

Oberon0

LDTA Tool Challenge

ABSTRACT

We describe an implementation of an Oberon0 compiler using the techniques proposed in the CoCoCo project. The compiler is constructed out of a collection of pre-compiled, statically type-checked language-definition fragments written in Haskell.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

As a case study of the techniques developed in the CoCoCo project,¹ we participated in the LDTA 2011 Tool Challenge.² The challenge was to implement a compiler for Oberon0, a small (Pascal-like) imperative language designed by Niklaus Wirth as an example language for his book on “Compiler Construction” [1].

The goal of the challenge is to contribute to “a better understanding, among tool developers and tool users, of relative strengths and weaknesses of different language processing tools, techniques, and formalisms”. The preface to this volume describes the challenge, which is divided into a set of incremental sub-problems. We have implemented all subproblems individually, and all the code is available from Hackage as the oberon0³ package.

CoCoCo comprises a set of libraries and tools in the form of a collection of embedded domain specific languages (EDSL) for constructing extensible compilers, where compilers are composed out of *separately compiled* language-definition fragments. The Haskell type system not only checks the expressions inside a component, but also the mutual consistency of a collection of components. Our approach builds on:

- the possibility to represent mutually dependent recursive structures, such as context free grammars, and to inspect and manipulate such structures in a type-safe way [2]
- the description of typed grammar fragments as first class Haskell values [3]

E-mail address: mviera@fing.edu.uy (M. Viera).

¹ Compositional Compiler Construction – <http://www.cs.uu.nl/wiki/bin/view/Center/CoCoCo>.

² <http://ldta.info/tool.html>.

³ <http://hackage.haskell.org/package/oberon0>.

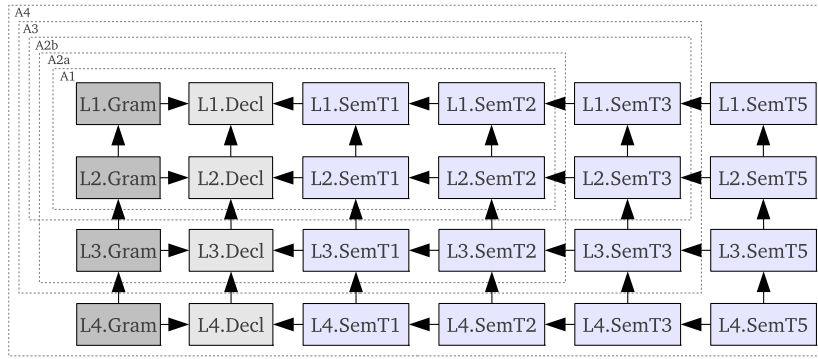


Fig. 1. Architecture of the Oberon0 implementation.

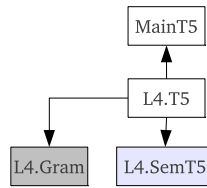


Fig. 2. Architecture of Artefact 4.

- the definition of a typed Left-Corner Transform (LCT), which removes left-recursion from a grammar; this enables us to use top-down parsing techniques to build parsers out of grammar fragments [4] and takes away many of the problems arising from composing grammar fragments into larger grammars.
- the possibility to construct a self-analysing, error correcting parser on the fly [5,6] without placing unreasonable constraints on the described language.
- the possibility to deal with attribute grammars as first class Haskell values, which can be transformed, composed and finally evaluated [7–9].

2. Architecture

The architecture of our implementation of Oberon0 is given in Fig. 1; each box corresponds to a separate Haskell module and arrows reflect the Haskell **import** relation (e.g. the module *L4.SemT5* imports *L4.SemT3* and *L3.SemT5*). Each module contains a collection of normal Haskell definitions and can be compiled separately. The design is incremental: each row corresponds to a syntactic extension (language levels) and each column corresponds to a semantic extension (task); each artefact in the challenge is represented by a dashed box containing the modules involved in that artefact. For each language level *L1* to *L4*:

- *Gram* modules contain syntax definition in the form of first-class grammar fragments; i.e. they define the context free grammar of the language(s)
- *Decl* modules contain the definition of the type of the semantics' record, and thus the interface to the abstract syntax of the language at hand
- *Sem* modules implement the semantics of each task in the form of rules which together constitute an attribute grammar

Notice that we do not include modules for Task 4. In Section 4.4 we explain how we get this task done almost for free by using attribute grammar macros when defining *L2*. The code for these macros is included in the semantics declaration module of *L2*, i.e. *L2.Decl*.

To build a compiler, e.g. Artefact 4 (Fig. 2), we import the syntax fragments (*l1t5*, *l2t5*, *l3t5* and *l4t5* from *L4.Gram*) and their respective semantics (*l1t5*, *l2t5*, *l3t5* and *l4t5* from *L4.SemT5*), combine them and build the compiler in the form of a parser which calls semantic functions. In Fig. 3 we show how to construct the parser of Artefact 4. The left-associative operator ($+>$) takes as its left-hand side operand a grammar and as its right-hand side operand a transformation to be applied. We start with an empty grammar (*emptyGram*) and apply the different language extensions to it. The function *closeGram* closes the constructed grammar and applies the *left-corner transform* in order to remove potentially introduced left-recursion; as a consequence a straightforward combinator-based top-down parser can be constructed from the transformed grammar. Then *generate kws* generates a parser integrated with the semantics for the language starting from the first non-terminal, where the list *kws* is a list of keywords extracted from the grammar description. This takes care of the problem that a string which is an identifier at a lower language level may become a keyword at a higher language level. The function *parse* finally parses

```

gl4t5 = closeGram $ emptyGram +>>
      l1 l1t5 +>>
      l2 l2t5 +>>
      l3 l3t5 +>>
      l4 l4t5
pA4 = (parse . generate kws) gl4t5

```

Fig. 3. A parser for Artefact 4.

```

l1 sf = proc _ → do
  rec
    modul ← addNT < ...
    ...
    ss     ← addNT < [| (pSeqStmt sf)
                        stmt
                        (pFoldr (pSeqStmt sf, pEmptyStmt sf)
                              (|| ";" stmt ||)) ||
    stmt   ← addNT < [| (pAssigStmt sf) ident ":" exp ||
    <|> [| (pIfStmt sf)
          "IF" cond
          (pFoldr (pCondStmtL.Cons sf, pCondStmtL.Nil sf)
                  (|| "ELSIF" cond ||))
          mbelse
          "END" ||
    <|> [| (pWhileStmt sf) "WHILE" exp "DO" ss "END" ||
    <|> [| (pEmptyStmt sf) ||
    cond   ← addNT < [| (pCondStmt sf) exp "THEN" ss ||
    mbelse ← addNT < pMaybe (pMaybeElseStmt.Nothing sf
                             , pMaybeElseStmt.Just sf)
                             (|| "ELSE" ss ||)
    exp     ← addNT < ...
    ...
    exportNTs < exportList modul $ export cs.Expression    exp
                                . export cs.StmtSeq         ss
                                . export cs.Statement        stmt
                                . export cs.MaybeElseStmt    mbelse
                                . ...

```

Fig. 4. Fragment of the concrete syntax specification of L1 (from *L1.Gram*).

the input program and returns its computed semantics. In the actual implementation of Oberon0 we construct scanner-less parsers using the `uu-parsinglib`⁴ parser combinator library.

In Section 3 we describe the implementation of the *Gram* and *Decl* modules, while in Section 4 we show how the semantics of the language(s) were implemented (in the *Sem* modules).

3. Syntax

Using our combinator library⁵ [3] we represent the concrete syntax of each language fragment as a Haskell value.

A fragment of the code constructing the internal representation of the CFG of the initial language L1 (module *L1.Gram*) is given in Fig. 4 using the Haskell *arrow* construct, which implicitly passes on the grammar under construction. The CFG fragments are written in so-called *idiomatic* style [10], where the Haskell function identifier *il* is formatted as `| |` and the constructor *li* is formatted as `| |`. Between these brackets we start by retrieving the semantic function for this production from the record *sf*; this function is used to combine the results of the parsing elements that make up the rest of the idiomatic

⁴ <http://hackage.haskell.org/package/uu-parsinglib>.

⁵ MUtually Recursive Definitions Explicitly Represented – <http://hackage.haskell.org/package/murder>.

```

data Statement = AssigStmt { id_AssigStmt  :: String
                           , exp_AssigStmt :: Expression }
               | IfStmt    { if_IfStmt    :: CondStmt
                           , elsif_IfStmt  :: CondStmtL
                           , else_IfStmt   :: MaybeElseStmt }
               | WhileStmt { exp_WhileStmt :: Expression
                           , ss_WhileStmt  :: Statement }
               | SeqStmt   { s1_SeqStmt   :: Statement
                           , s2_SeqStmt   :: Statement }
               | EmptyStmt

type CondStmtL = [CondStmt]
data CondStmt  = CondStmt { exp_CondStmt :: Expression
                           , ss_CondStmt  :: Statement }
type MaybeElseStmt = Maybe Statement
data Expression = ...

```

Fig. 5. Abstract syntax of the statements of L1 (from *L1.Decl*).

construct into the result of this construct. We use the `murder` combinators `pFoldr` and `pMaybe` to model repetition and option, respectively. These combinators are analogous to the respective `foldr` and `maybe` functions.

The parameter `sf` is a record containing the “semantics of the complete language”; its type is defined in the module *L1.Decl* and is derived from the abstract syntax of which we show a fragment in Fig. 5. Thus the syntax definition of the language is parametric on its semantics. The semantics can be defined separately (in the *Sem* modules) and extended without having to change the code defining the syntax.

We use the Template Haskell function `deriveLang`⁶ to derive the type of the record (the `sf` parameter) from the list of data types (abstract non-terminals) which together are used to represent abstract syntax trees. For the example fragment we call:

```

$(deriveLang "L1" ["Module", "Statement", "Expression",
                  "CondStmtL", "CondStmt", "MaybeElseStmt"])

```

For each production of the abstract syntax tree the produced record contains a field with as name the name of the production prefixed by a `p` and as type the type of the semantic function, which is defined in terms of the semantics associated with the children of the production. For example, the field generated for the production *AssigStmt* is:

```

pAssigStmt :: sf_id_AssigStmt → sf_exp_AssigStmt → sf_AssigStmt

```

where `sf_id_AssigStmt`, `sf_exp_AssigStmt` and `sf_AssigStmt` are type variables corresponding to the types of the semantic values associated to identifiers, expressions and assignments, respectively.

For the cases of *List* or *Maybe* type aliases, fields are produced using the name of the non-terminal (i.e. the type) to disambiguate. For example, for *CondStmtL* we generate the fields `pCondStmtL_Cons` and `pCondStmtL_Nil`, and for *MaybeElseStmt* we generate the fields `pMaybeElseStmt_Just` and `pMaybeElseStmt_Nothing`.

Grammars defined in this way are *extensible*, since other modules may define further transformations to be applied to the grammar under construction. Each grammar transformation exports (with *exportNTs*) its starting point (e.g. *modul*) and a table of *exported non-terminals*, each consisting of a label (by convention of the form `cs_...`) and a reference to the current definition of that non-terminal, again a plain Haskell value which can be used and modified in future extensions. Fig. 6 contains a fragment of the definition of L2 (from module *L2.Gram*), which extends the L1 grammar with a **FOR**-loop. Again the code makes use of the Haskell *arrow* construct. We start by retrieving references to all current representations of non-terminals which are to be extended or used (using *getNT*) from the *imported* non-terminals. Next we add new productions to existing non-terminals with *addProds*; this does not introduce any references to new non-terminals. New non-terminals can still be introduced as well using *addNT*. The Haskell type-system ensures that the *imported* list indeed contains a table with entries `cs_StmtSeq`, `cs_Statement`, `cs_Expression` and `cs_Ident`, and that the types of these non-terminals coincide with their use in the semantic functions of the extensions.

The definition in Fig. 6 may look a bit verbose. This is caused by the interface having been made explicit. Using some Template Haskell this can easily be overcome.

⁶ Provided by the package *AspectAG*.

```

l2 sf = proc imported → do
  let ss    = getNT cs_StmtSeq    imported
  let stmt = getNT cs_Statement  imported
  let exp  = getNT cs_Expression imported
  let ident = getNT cs_Ident      imported
  ...
  rec
    addProds <- (stmt    ,  || (pForStmt sf) "FOR" ident ":@" exp dir exp mbeexp
                          "DO" ss "END" ||)

    dir      <- addNT <- || (pTo sf) "TO" || <|> || (pDownto sf) "DOWNTO" ||
    mbeexp   <- addNT <- pMaybe (pCst1Exp sf, id) (|| "BY" exp ||)
    ...
  exportNTs <- imported

```

Fig. 6. Fragment of the grammar extension L2 (from *L2.Gram*).

```

data EXT_Statement
  = ForStmt { id_ForStmt :: String, start_ForStmt :: Expression
            , dir_ForStmt :: ForDir, stop_ForStmt :: Expression
            , step_ForStmt :: Expression, ss_ForStmt :: Statement }
  | ...
data ForDir = To | Downto
data EXT_Expression = Cst1Exp
...

```

Fig. 7. Abstract syntax of the **FOR**-loop of L2 (from *L2.Decl*).

Fig. 7 shows the abstract syntax fragment corresponding to the **FOR**-loop extension. The prefix *EXT_* indicates that this definition is extending a given non-terminal, which is again consumed by some Template Haskell to generate the new definitions.

4. Aspect oriented semantics

The semantics of Oberon0 were implemented using the `AspectAG`⁷ library [7] which embeds attribute grammars in Haskell. By using attribute grammar systems, language semantics can be described in an aspect oriented way [11,12]; i.e. an evaluator is assembled from a collection of attribute grammar fragments, each describing a specific aspect of the language.

An attribute grammar is a context-free grammar of which non-terminals are decorated with values, called attributes. To compute an attribute in a production, the values of the attributes of the parent and the children can be used. We distinguish two kind of attributes: synthesised and inherited attributes. The synthesised attributes are passed up to the parent nodes, while the inherited attributes are passed down from the parent.

In order to be able to redefine attributes or to add new attributes later, we encode the lists of inherited and synthesised attributes of a non-terminal as an `HList`-encoded [13] value; with each attribute we associate a unique type which is used as an index in such a “list”. The lookup process is performed by the Haskell class mechanism. In this way the *closure test* of the attribute grammar (each attribute has a single definition) is implicitly realised by the Haskell compiler when building the right instances of the classes. Thus, attribute grammar fragments can be individually type-checked, compiled, distributed and composed to construct a compiler.

4.1. Pretty-printing

Pretty-printing is implemented by a synthesised attribute, called *spp*,⁸ using the pretty-printing combinators provided by the `uulib`⁹ package.

⁷ <http://hackage.haskell.org/package/AspectAG>.

⁸ By convention we use the prefix *i* and *s* for inherited and synthesised attributes, respectively.

⁹ <http://hackage.haskell.org/package/uulib>.

Attributes have to be defined for each production of the abstract syntax tree. For example, the following code defines how the attribute *spp* is to be computed for the production *CstDecl* which declares a constant.

```
sppCstDecl = syn spp $ do
  nm ← at ch_id_CstDecl
  exp ← at ch_exp_CstDecl
  return $ value nm >#< " = " >#< exp # spp >#< " ; "
```

The production has two children: *ch_id_CstDecl*, of type (*DTerm String*),¹⁰ and *ch_exp_CstDecl*, an expression node. The *at* function gives us access to the attributes associated with a node of the production. The *besides* combinator (*>#<*) is used to combine the string value of *ch_id_CstDecl*, the string "=", the pretty-printed value of the child *ch_exp_CstDecl* and the string ";".

We construct the semantic functions for the productions by applying the *knit* function to the (composed) aspects of a given task. This function connects the local attribute flow of the production, as described by the composed aspects, with the flows from the inherited to the synthesised attributes of each child and constructs the flow from the inherited attributes of the root node to its synthesised attributes. In the case of Task 1 the aspects of the production *CstDecl* consist of the single attribute *spp*.

```
aspCstDecl = sppCstDecl
```

4.2. Name analysis

Error messages produced by the name analysis are collected in a synthesised attribute *serr*. The default behaviour of this attribute for most productions is to concatenate the errors produced by its children. To capture this pattern we introduce a generic rule *serrRule* computing this attribute using the function *use* from the *AspectAG* library, which takes as arguments the label of the attribute to be defined (*serr*), the Haskell list of non-terminals (labels) for which the attribute is to be defined (*serrNTs*), an operator telling how to combine the attribute values (*++*), and a unit value to be used when none of the children has such an attribute (*[] :: [String]*).

```
serrRule = use serr serrNTs (++) ([] :: [String])
```

When a new name is defined we check for multiple declarations and at name uses we check for incorrect uses or uses of undefined identifiers, producing error messages when appropriate. The code below shows the definition of *serr* for the use of an identifier represented by a production *IdExp*, which has a child named *ch_id_IdExp* of type (*DTerm String*).

```
serrIdExp = syn serr $ do
  lhs ← at lhs
  nm ← at ch_id_IdExp
  return $ checkName nm (lhs # ienv) [ "Var", "Cst" ] "an expression"
```

With the (plain Haskell) function *checkName* we lookup the name (*nm*) in the symbol table (inherited attribute *ienv* coming from the left-hand side) and, if it is defined, we verify that the name represents either a variable ("Var") or a constant ("Cst") and generate a proper error message if not.

Note that the function *use* is just a plain Haskell function, which constructs (a boiler plate) part of the language definition at hand. We consider this one advantage of our embedded approach; common semantic patterns can directly be supported by defining additional supporting functions.

The symbol table is implemented by the pair of attributes *senv* and *ienv*. The synthesised attribute *senv* collects the information from the name declarations and the inherited attribute *ienv* distributes this information through the tree.

In order to perform the name analysis, the type of the symbol table could have been *Map String NameDef*, which is a map from names to values of type *NameDef* representing information about the bound name. However, since we want to use the same symbol table for future extensions, we keep the type “non-closed” by using a list-like structure:

```
data SymbolInfo b a = SI b a
type NMap a = Map String (SymbolInfo NameDef a)
```

For the current task the symbol table includes values of type *NMap a*, parametric in *a*, where *a* represents the “any other information we might want to store for this symbol in the future”. In the example below, for declarations of constants, the

¹⁰ *DTerm a* is the type used by *murder* to represent *attributed terminals* (i.e. identifiers, values); it encodes the value (*value*) and position in the source code (*pos*) of the terminal.

table consists of a map from the introduced name to a *SymbolInfo* which (at least) includes the information needed by the name analysis (constructed using *cstDef*):

```
senvCstDecl r = syn senv $ do
  nm ← at ch_id_CstDecl
  return $ Map.singleton (value nm) (SI (cstDef $ pos nm) r)
```

As easily as we introduced *use* for dealing with the default cases of synthesised attributes, we can capture the behaviour of distributing an inherited attribute to all the children of a production with the function *copy* from the *AspectAG* library:

```
ienvRule _ = copy ienv ienvNTs
```

The various aspects introduced by the attributes are combined using the function *ext*:

```
aspCstDecl r = senvCstDecl r `ext` ienvCstDecl r `ext` serrCstDecl `ext`
  T1.aspCstDecl
```

In this case, for the production *CstDecl*, we extend *T1.aspCstDecl*, which is imported from *L1.SemT1* and includes the pretty-printing attribute, with the attributes implementing the name analysis task (*serr*, *ienv* and *senv*).

Once the attributes definitions are composed, the semantic functions for the productions may be computed using the function *knit*. For example, the semantic function of the production *CstDecl* in the case of *L1.SemT2* is *knit (aspCstDecl ())*. The use of *()* (unit) here is just to “close the symbol table”, since no further information is to be stored for the rules of Task 2.

4.3. Type checking

Type error messages are collected in the synthesised attribute *sterr*. For type checking we extend the symbol table with the type information (*TInfo*) of the declared names. This is done by *updating* the computation of the attribute *senv* with the function *synupdM*, which is similar to *syn* but redefines it making use of its current definition. In the following example we update the symbol table information for the production *VarDecl*, where *sty* is an attribute defined for expressions and types, computing their type information:

```
senvVarDecl' r = synupdM senv $ do
  typ ← at ch_typ_VarDecl
  return $ Map.map (λ(SI nd _) → (SI nd $ SI (typ # sty) r))
```

The parameters to *synupdM* are *senv*, the attribute to update, and a monadic computation that returns a function from the previously defined value of the attribute to the new one. We capture the previously computed symbol table with the pattern *(SI nd _)* and extend it with the type information *(typ # sty)* of the variable. The previous definition of the type information is just ignored and only used to indicate the type of the symbol table. Thus, thanks to lazy evaluation, when extending the aspects of Task 2 we only need to pass an undefined value of type *SymbolInfo TInfo a*, where *a* is the type of the new rest of the symbol table (for future extensions):

```
undTInfo :: a → SymbolInfo TInfo a
undTInfo = const ⊥
aspVarDecl r = (senvVarDecl' r)
  `ext` sterrRule
  `ext` (T2.aspVarDecl $ undTInfo r)
```

One of the problems we encountered is that new tasks place new requirements on the type information we want to maintain. Unfortunately Haskell does not have so-called open data types, which would have come in handy here. Instead we decided to resort to the use of the type *Dynamic*. A *TInfo*, with the information associated with a specific type, consists of: the representation *trep* of the given type, encapsulated as a *Dynamic* value, a *String* with its pretty-printing (*tshow*), and a function *teq* that can be used to relate the stored type information with type information coming from elsewhere.

```
data TInfo = TInfo { trep :: Dynamic
  , tshow :: String
  , teq :: (TInfo → Bool) }
```

The main task of type checking is to verify whether the actual type of an expression is compatible with the type expected by the context of that expression, e.g. whether the condition of an **IF** statement has type *BOOLEAN*.

```

check pos expected got
= if (expected 'teq' got) ∨ (got 'teq' unkTy) ∨ (expected 'teq' unkTy)
  then []
  else [show pos ++ ": Type error. Expected " ++ show expected ++
        ", but got " ++ show got]

```

If either the expected or the obtained type is unknown (*unkTy*) we do not report a type error, because unknown types are generated by errors that have been already detected by the name analysis process.

A very simple case of type information is the elementary type `BOOLEAN`, where we do not provide any other information than the type itself. Thus, the type representation is implemented with a singleton type *BoolType*.

```

data BoolType = BoolType
boolTy = let d = toDyn BoolType
         bEq = (≡) (dynTypeRep d) . dynTypeRep . trep . baseType
         in TInfo d "BOOLEAN" bEq

```

To construct the corresponding *TInfo* we convert a *BoolType* value into a *Dynamic* with the function *toDyn*. A type is compatible with `BOOLEAN` if its base type¹¹ is also `BOOLEAN`, i.e. is compatible if both types are represented with *BoolType* values. With the function *dynTypeRep* we extract a concrete representation of the type of the value inside a *Dynamic* that provides support for equality.

There exist some other cases where a more involved type representation is needed. For example, in the case of `ARRAY` we include the type information of its elements and the length of the array, if it can be statically computed.

```

data ArrType = ArrType (Maybe Int) TInfo

```

Then, by using the type-safe cast function *fromDynamic* we can get access to this information provided the dynamic typed value represents an array. In this way we can, when trying to index a variable, check whether the index is out of range; in case the cast does not succeed we indicate that the variable we are trying to access is not an array:

```

checkSelArray pos ty ind
= case (fromDynamic . trep . baseType) ty of
  Just (ArrType l _) → checkIndex pos ind l
  _                 → [show pos ++ ": Accessed variable is not an array"]

```

We use the same technique to store information about the fields of a `RECORD` and the parameters of a `PROCEDURE`.

4.4. Source-to-source transformation

In [9] we extended *AspectAG* with an *agMacro* combinator that enables us to define the attribute computations of a new production in terms of the attribute computations of existing productions. We defined the semantics of the extensions of the language level L2 using this macro mechanism. The `FOR`-loop is implemented as a `WHILE`-loop and the `CASE` statement is defined in terms of an `IF-ELSIIF-ELSE` cascade.

Fig. 8 contains the macro definition for the `FOR`-loop, which is parametrised by the attributes (semantics) of:

- *SeqStmt*: sequence of statements
- *AssigStmt*: assign statement
- *WhileStmt*: while statement
- *IntCmpExp*: integer comparison expression
- *IdExp*: identifier expression
- *IntBOpExp*: integer binary operation expression

We use the combinator *withChildAtt* to obtain the value of the *self* attribute of the child *ch_dir_ForStmt*, with the direction of the iteration. In case the value is *To* the loop counter is incremented (*Plus*) on each step while is less or equal (*LECmp*) the stop value. In other case (*Downto*) we use *Minus* to decrement the counter and *GECmp* (greater or equal) to compare it with the stop value.

An attribute grammar macro (*agMacro*) takes as a parameter the rule that is used as a basis (called base rule) and the mappings of the children of the base rule to the children of the new macro rule. For example, in the following fragment the

¹¹ In case of a user type, the type it denotes.

```

macroForStmt aspSeqStmt aspAssigStmt aspWhileStmt
             aspIntCmpExp aspIdExp aspIntBOpExp
= withChildAtt ch_dir_ForStmt self $ λdir →
let (op_stop, op_step) = case dir of
    To      → (LECmp, Plus)
    Downto  → (GECmp, Minus)

initStmt  = (aspAssigStmt , ch_id_AssigStmt  ↦ ch_id_ForStmt
             <.> ch_exp_AssigStmt ↦ ch_start_ForStmt)

whileStmt = (aspWhileStmt , ch_exp_WhileStmt ⇒ condWhile
             <.> ch_ss_WhileStmt ⇒ bodyWhile)

condWhile = (aspIntCmpExp , ch_op_IntCmpExp ↦ op_stop
             <.> ch_e1_IntCmpExp ⇒ idExp
             <.> ch_e2_IntCmpExp ↦ ch_stop_ForStmt)

idExp     = (aspIdExp      , ch_id_IdExp     ↦ ch_id_ForStmt)

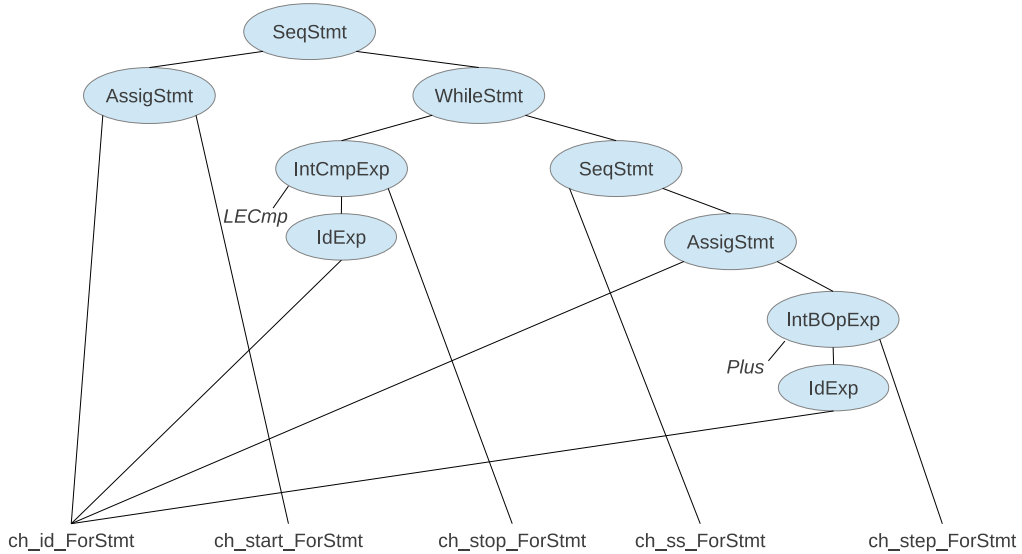
bodyWhile = (aspSeqStmt   , ch_s1_SeqStmt   ↦ ch_ss_ForStmt
             <.> ch_s2_SeqStmt ⇒ stepWhile)

stepWhile = (aspAssigStmt , ch_id_AssigStmt ↦ ch_id_ForStmt
             <.> ch_exp_AssigStmt ⇒ expStep)

expStep   = (aspIntBOpExp , ch_op_IntBOpExp ↦ op_step
             <.> ch_e1_IntBOpExp ⇒ idExp
             <.> ch_e2_IntBOpExp ↦ ch_step_ForStmt)

in withoutChild ch_dir_ForStmt
   (agMacro (aspSeqStmt , ch_s1_SeqStmt ⇒ initStmt
               <.> ch_s2_SeqStmt ⇒ whileStmt))

```

Fig. 8. Macro definition of the **FOR**-loop.Fig. 9. The **FOR**-loop in terms of the original AST.

semantics of the sequence (*aspSeqStmt*) is used as the base rule:

```

(agMacro (aspSeqStmt , ch_s1_SeqStmt ⇒ initStmt
          <.> ch_s2_SeqStmt ⇒ whileStmt))

```

The children mapping definitions are composed with the (*<.>*) combinator. The (*⇒*) combinator maps a child to another sub-macro; thus, in the code above the children of the sequence rule *ch_s1_SeqStmt* and *ch_s2_SeqStmt* are mapped to the sub-macros defined by *initStmt* and *whileStmt*, respectively.

We use the combinator (\hookrightarrow) to map a child of the body rule to a child of the macro rule, and the combinator (\rightsquigarrow) to define a mapping from a child to a literal value. In Fig. 9 we show the structure of the macro (i.e. the **FOR**-loop in terms of the original AST) for the *To* case.

That can be seen as a code translation from:

```
FOR id := start TO stop BY step DO
  ss
END

to:

id := start;
WHILE id <= stop DO
  ss;
  id := id + step
END
```

In the cases where specialised behaviour is needed, like for example pretty-printing, it is still possible to redefine the attributes involved on these aspects. Thus, the following is a fragment of the pretty-printing of **FOR**, where *synmodM* is similar to *syn* but instead of defining a new attribute replaces an existing definition when extending a semantics that contains it:

```
sppForStmt
= synmodM spp $ do id  <- at ch_id_ForStmt
                  start <- at ch_start_ForStmt
                  dir   <- at ch_dir_ForStmt
                  stop  <- at ch_stop_ForStmt
                  step  <- at ch_step_ForStmt
                  ss    <- at ch_ss_ForStmt
                  return $ "FOR" >#< value id >#< " := " >#<
                        start # spp >#< dir # spp >#< stop # spp >#<
                        "BY" >#< step # spp >#<
                        pp_block " DO " " END " " ; " (ss # sppl)
```

As such, our mechanism is much more expressive than a conventional macro mechanism, which only performs a structure transformation. Using the library we get Task 4 almost for free. However, it should be noticed that we showed a simplification of the **FOR** semantics that does not capture the upper bound of the loop. For example, the following fragment loops infinitely:

```
n := 10;
FOR i := 0 TO n DO
  n := n + 1
END
```

To solve this problem we extended the target language to allow the definition of scopes and local variables, and declared a scope for each **FOR** with a local variable that captures this bound.

Our approach is not very suitable for some other kind of source-to-source transformations like optimisations, because we do not represent the AST with values (if we want to keep the AST extensible) and we (still) do not have higher-order attributes. A possible approach is however to generate an AST of a fixed core language and perform the optimisations on this representation.

4.5. Code generation

We generate the C abstract syntax representation provided by the package *language-c*.¹² This package also includes a pretty-printing function for the abstract syntax. Since ANSI C does not allow for nested functions we have to lift all the procedures, types and constants definitions to top-level when generating the C code required by the challenge (note that the lifting as specified is trivial, since nested procedures in Oberon0 cannot refer to variables defined in the outer scope, and hence the exercise does not require bindings to be properly lifted). In order to avoid name clashes with C keywords or as a result of the lifting process, we rename every identifier to make it unique. New names are composed by: a character `'_'` (assuring no clashes with C keywords), the path (module and procedure names) to the scope where the name is defined

¹² <http://hackage.haskell.org/package/language-c>.

Table 1

Code sizes (in lines of code) of the components of the compiler.

Lang./Task	Common	T1	T2	T3	T5	Total
Common	–	42	14	–	23	79
L1	128	156	147	220	228	879
L2	187	98	69	65	56	475
L3	94	75	75	134	145	523
L4	48	67	56	197	95	463
Total	457	438	361	616	547	2419

and the actual name. Thus, if we have the following Oberon0 program:

```

MODULE A;
  VAR BC : INTEGER;
  PROCEDURE B;
    PROCEDURE C;
      END C
    END B
  END A.

```

The names are mapped: the variable name `BC` to `A_BC`, the procedure name `B` to `A_B` and the procedure name `C` to `A_B_C`. Since underscore is not allowed in Oberon0 identifiers, this renaming does not introduce new clashes, like the one we could have had with `C` if the variable `BC` was called `B_C`.

To implement the renaming we extend the symbol table with the name mapping. Thus, for example, when defining the code generation attribute for the production *IdExp* (use of identifiers) the new name is located in the table.

5. Artefacts

In Table 1 we show the size (in lines of code without comments) of our implementations, disaggregated into the different tasks and language levels. The *Common* column includes the *Gram* and *Decl* files, while the *Common* row includes some code used by the *Main* modules. The code includes 26 lines of Template Haskell, calling functions defined in the libraries to avoid some boilerplate.

We have implemented all the combinations from L1-T1 to L4-T5, including the artefacts proposed by the challenge.

6. Conclusions

The most important aspect of our approach is the possibility to construct a compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments written in Haskell, but with a DSL taste.

When looking at all the aspects we have covered, we can conclude that we managed to find solutions for all of them. This should not come as a surprise since we could always fall back to plain Haskell in those cases where our libraries were not providing a standard solution for the problem at hand. We have seen such solutions for dealing with flexible symbol tables, generating new identifiers and types.

We mention again that our implementation is somewhat verbose, since each module contains quite some code “describing its interface” in the collection of co-operating modules. This is the price we have to pay for getting the extreme degree of flexibility we are providing. By collapsing the modules the amount of linking information shrinks considerably. We can reduce verbosity even further by using the conventional attribute grammar system *uuagc* to generate *AspectAG* code [8].

Another cause of the verbosity is that we have not used the system itself nor Template Haskell to capture common patterns. We have chosen to reveal the underlying mechanisms, the role of the type system, the full flexibility provided, and have left open the possibility for further extensions.

The *expression problem* [14] is the problem of being able to extend a program (without recompiling existing code) both by adding cases to its data types and new functions over these datatypes. In functional programming is easy to add new functions, but data types are fixed. This lack of extensible data types makes it not entirely straightforward to use our techniques to implement AST transformations for extensible languages. Semantic macros solve some of these problems. A possible approach is to use our technique to implement the front-end of a compiler, translating to a core fixed language, and then resort to other, more traditional approaches (like *uuagc*) to implement the back-end; since *uuagc* is completely text-based and always generates the text of a complete compiler the aforementioned problem does not arise. Another option is to use *data types à la carte* [15] or other class-based approaches [16] to tackle the expression problem in Haskell.

References

- [1] N. Wirth, *Compiler Construction*, International Computer Science Series, Addison-Wesley, 1996.
- [2] A.I. Baars, S.D. Swierstra, M. Viera, Typed transformations of typed abstract syntax, in: TLDI '09: Fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation, ACM, New York, NY, USA, 2009, pp. 15–26.

- [3] M. Viera, S.D. Swierstra, A. Dijkstra, Grammar fragments fly first-class, in: *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, 2012, pp. 47–60.
- [4] A.I. Baars, S.D. Swierstra, M. Viera, Typed transformations of typed grammars: the left corner transform, in: *Proc. of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, 2009, pp. 18–33.
- [5] S.D. Swierstra, Parser combinators: from toys to tools, in: G. Hutton (Ed.), *Haskell Workshop*, 2000.
- [6] S.D. Swierstra, Combinator parsers: a short tutorial, in: A. Bove, L. Barbosa, A. Pardo, J. Sousa Pinto (Eds.), *Language Engineering and Rigorous Software Development*, in: LNCS, vol. 5520, Springer, 2009, pp. 252–300.
- [7] M. Viera, S.D. Swierstra, W. Swierstra, Attribute grammars fly first-class: how to do aspect oriented programming in Haskell, in: *Proc. of the 14th Int. Conf. on Functional Programming*, ACM, New York, USA, 2009, pp. 245–256.
- [8] M. Viera, S.D. Swierstra, A. Middelkoop, UUAG meets AspectAG, in: *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, 2012.
- [9] M. Viera, S.D. Swierstra, Attribute grammar macros, *Sci. Comput. Program.* 96 (2) (2014) 211–229.
- [10] C. McBride, R. Paterson, Applicative programming with effects, *J. Funct. Program.* 18 (1) (2008) 1–13.
- [11] U. Kastens, W.M. Waite, Modularity and reusability in attribute grammars, *Acta Inform.* 31 (7) (1994) 601–627.
- [12] R. Lämmel, Declarative aspect-oriented programming, in: O. Danvy (Ed.), *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, Texas, USA, January 22–23, 1999, Technical report BRICS-NS-99-1, University of Aarhus, 1999, pp. 131–146.
- [13] O. Kiselyov, R. Lämmel, K. Schupke, Strongly typed heterogeneous collections, in: *Proc. of the 2004 Workshop on Haskell*, ACM Press, 2004, pp. 96–107.
- [14] P. Wadler, The expression problem, e-mail available online <http://www.daimi.au.dk/~madst/tool/papers/expression.txt>, 1998.
- [15] W. Swierstra, Data types à la carte, *J. Funct. Program.* 18 (4) (2008) 423–436.
- [16] R. Lämmel, K. Ostermann, Software extension and integration with type classes, in: S. Jarzabek, D.C. Schmidt, T.L. Veldhuizen (Eds.), *Generative Programming and Component Engineering*, *Proceedings of the 5th International Conference, GPCE 2006*, Portland, Oregon, USA, October 22–26, 2006, ACM, 2006, pp. 161–170.