Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (5) (2011) 47–69
www.elsevier.com/locate/entcs

Visitor-based Attribute Grammars with Side Effect

Arie Middelkoop[1] Atze Dijkstra[1] S. Doaitse Swierstra[1]

Universiteit Utrecht, The Netherlands

Abstract

The visitor design pattern is often applied to program traversal algorithms over Abstract Syntax Trees (ASTs). It defines a visitor, an object with a visit method that is executed for each node in the AST. These visitors have the advantage that the order of traversal is explicitly under control of the programmer, which is essential to deal with side-effectful computations. Unfortunately, the exchange of results between traversals is error-prone.
Attribute Grammars (AGs) are an alternative way to write multi-traversal algorithms. An attribute evaluator decorates the AST with attributes in one or more traversals. The attributes form a convenient mechanism to exchange results between traversals. Unfortunately, AGs discourage the use of side effect.
In this paper, we present RULER-FRONT, a language capturing the combination of the above approaches. A RULER-FRONT grammar can be translated to traversal algorithms in multiple languages. In this paper, we translate to the imperative, dynamically-typed language JAVASCRIPT.

Keywords: attribute grammar, visitor, design pattern

1 Introduction

Algorithms for traversing tree-shaped data structures appear in many applications, especially in compilers. A lot of effort has been invested in proper abstractions for tree traversals, for example in the form of Attribute Grammars (AGs) [11]. In the last years, we applied AGs in many small projects (to teach compiler construction [21], master projects, etc.), and several large projects, including the Utrecht Haskell Compiler [4], the Helium [8] compiler for learning Haskell, and the editor Proxima [19]. The use of AGs in these projects is invaluable, for reasons that become clear in Section 2.

Tree traversals play their role in many other fields, including end-user applications. Web applications, for example, traverse and compute properties of DOM trees. Sadly, the nice abstractions emerging from the compiler field are not used to write such traversals. One of these reasons is that AGs require an additional

[1] Email: {ariem,atze,doaitse}@cs.uu.nl

doi:10.1016/j.entcs.2011.06.004

language to be learned. Also, the AG formalism poses too severe restrictions to be used effectively in these areas, such as prohibition of side effect, or tool support may simply be absent for the programming language in question. The purpose of this paper and associated work is to treat the above two technical challenges.

Considering the first challenge, for imperative languages like JavaScript, a programmer either writes recursive functions, or takes a more structured approach via the visitor design pattern [7,17,16]. Tool support for the visitor design pattern is available for many languages. For example, the parser generator SableCC [6] generates visitor skeleton code for the ASTs that the parser produces, and we used these once to write a type checker for MiniJava [18]. We also used ASM [3], a library used in many big Java projects that provides visitor skeleton code to traverse Java bytecode, to transactify Java programs [1]. With visitors, we use side-effect to carry results computed in one visit over to the next. In our experience, scheduling visits and side effect is an error-prone process, due to absence of the define-before-use guarantee. We elaborate on this in Section 2.1.

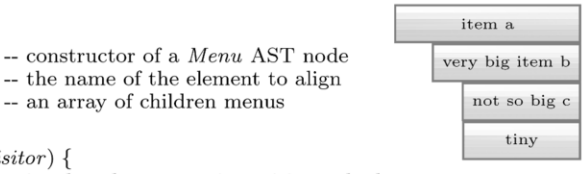
Attribute grammars offer a programming model where each AST node has attributes (named values per node). The programmer writes code that computes attributes in terms of other attributes. The attribute grammar evaluator automatically schedules this code over visits, and define-before-use can be verified with the circularity test of AGs. The implicitness of scheduling is a serious advantage, because it saves us from writing this scheduling manually, and cannot do it wrong. Unfortunately, the implicitness of scheduling comes with a severe restriction: side effect cannot be used reliably and should not be used in attribute computations. In web applications, for example, we typically would like to use a bit of side effect to influence the contents of a webpage. We elaborate on this in Section 2.2.

The main contribution of this paper is an extension of attribute grammars that has an explicit notion of visits, which offers a hybrid model between visitors and attribute grammars, while maintaining the best of both worlds. In fact, besides being more expressive, our extension make attribute grammars more intuitive to use.

To accomplish this goal, we also address the second challenge, which is to make our approach available for many target languages. We present ruler-front, a small but powerful language for tree traversals. We managed to isolate the language-dependent part into a small subset called ruler-core, and show the translation from ruler-core to JavaScript. In a related paper [14], we showed a translation to Haskell. With these two languages, we cover the implementation issues regarding the full spectrum of mainstream general purpose programming languages available today.

Similar to Yacc, SableCC and UUAG [20], the idea is to embed code fragments of the target language for the computations of attributes. This keeps general-purpose programming constructs out of ruler-core, and allows the programmer to express computations without having to learn a special language. In particular, ruler-core is suitable as a target language for attribute grammars.

In summary, we present two languages ruler-front and ruler-core. We im-




```
function Menu (name, children) {        -- constructor of a Menu AST node
  this.name     = name;                  -- the name of the element to align
  this.children = children;              -- an array of children menus
}
Menu.prototype.accept = function (visitor) {
  visitor.visitMenu (this);              -- invokes the appropriate visit method
}
function Visitor () {                    -- constructor of a Visitor object
  this.depth   = 0;                      -- the depth so far in the menu tree
  this.maximum = 0;                      -- the maximum width observed so far
  this.count   = 0;                      -- the number of menus laid out so far
}
var root =                               -- the menu tree and corresponding html nodes
  new Menu ("a", [                       -- <div id="a">item a</div>
    new Menu ("b", [                     -- <div id="b">very big item b</div>
      new Menu ("c", [])                 -- <div id="c">not so big c</div>
    , new Menu ("d", [])                 -- <div id="d">tiny</div>
    ])
  ]);    -- <div id="anchor" onLoad="align(root,this);"></div>
function align (root, anchor) {          -- aligns the html nodes according to the menu tree
  var v = new Visitor ();                -- creates visitor with empty state
  v.visitMenu = function (menu) {        -- first visit method (gets menu node as param)
    menu.elem  = document.getElementById (menu.name);
    menu.depth = this.depth;             -- remember depth for the second visit
    this.maximum = Math.max (this.maximum, this.depth * 20 + menu.elem.clientWidth);
    for (var i in menu.children) {
      this.depth = menu.depth + 1;       -- reset this.depth to one deeper than current
      menu.children [i].accept (this);   -- invokes visitor on children
    }
  }
  root.accept (v);                       -- invokes the first visit (on the root)
  v.visitMenu = function (menu) {        -- second visit method (gets menu node as param)
    var offset = menu.depth * 20;
    menu.elem.style.left   = (anchor.offsetLeft + offset) + "px";
    menu.elem.style.top    = (anchor.offsetTop + this.count * 30) + "px";
    menu.elem.style.width  = (this.maximum - offset) + "px";
    menu.elem.style.height = 30 + "px";
    this.count ++;                       -- inorder numbering of nodes
    for (var i in menu.children) {       -- invokes visitor on menus children
      menu.children [i].accept (this);   -- count should not be reset in this case
    }
  }
  root.accept (v);                       -- invokes the second visit (on the root)
}
```

Fig. 1: Pseudocode dualvisit menu alignment.

plemented both in a single tool written in Haskell using UUAG [2] . In Section 2 we investigate the above challenges in more detail. In Section 3 we present RULER-CORE, with a translation to JAVASCRIPT in Section 4. In the extended version of this paper [15], we give a translation from RULER-FRONT to RULER-CORE.

[2] Downloadable from svn: https://subversion.cs.uu.nl/repos/project.ruler.systems/ruler-core/

2 Example

In this section, we motivate the claims of the introduction in more detail, and introduce the background information relevant for the remainder of the paper. We take as usecase the alignment of an HTML menu in a web application using JavaScript, based on a multi-visit tree traversal over an abstract description of the menu. We first show a solution using the visitor-pattern, then a near-solution using attribute grammars, finally followed by two solutions using ruler-front.

2.1 Visitor design pattern.

In the visitor design pattern, each node of the Abstract Syntax Tree (AST) is modelled as an object, which stores references to the subtrees, and has an *accept* method. The *accept* method takes a visitor as parameter. A visitor is an object with a *visit*-method for each type of node. The *accept* method of the AST node calls the appropriate *visit*-method on the visitor and passes the node as an argument. This *visit* method consists of statements that manipulate the state of the visitor or the AST node, and can visit a subtree by calling the *accept* method on the root of a subtree, with the visitor-object as parameter.

Figure 1 shows an example of a visitor that layouts HTML items as a menu in a tree-like fashion, as visualized in the upper-right corner. The menus are aligned to the right, and submenus are slightly indented. Furthermore, we desire the smallest layout, based on the contents of the HTML items. The variable *root* contains an abstract description of the menu as a tree of *Menu* objects (the AST). Associated with each *Menu* object is an HTML item with the same name. We interpret the menu structure to layout the HTML items. In the first visit to the menu tree, we query the widths of the corresponding HTML items. In the second visit, we adjust the positions and sizes of these items. Some information (such as indentation based on the *depth*) is computed in the first visit, and also needed in the second visit. That information we store as additional fields in the menu objects.

The order in which the tree is visited is clearly defined by the explicit *accept*-calls in the *visit*-methods. This is important to deal with side effect: we need to have queried all the sizes of the HTML items before we start resizing them.

However, there are a number of issues with the above solution. In the second visit, we require a number of values computed in the first visit. We store these in the state of the AST nodes during the first visit. However, there is no guarantee that we actually stored them there in the first visit. Furthermore, we never remove any of these values from the state, and thus retain all memory until the AST gets deallocated. This especially becomes a problem when using large AST storing many results.

Furthermore, we have to take care of the order of the statements. For example, the *this.depth* needs to be reset at the appropriate place, and requires that the assignment to *menu.depth* is done before. Similarly, the increment to *this.count* needs to be positioned carefully. These are actually separate aspects which we would like to implement in isolation, without having to worry about their composition.

Finally, we need to explicitly write visits to children using *accept*. Some tools generate depth-first visitors, which alleviates the need to do so, but these come with restrictions. For example, all statements must be written before the invocations to children. In Figure 1 we reset *this.depth* in between visits to children. To use a depth-first visitor, we would have to move this statement (which may not be easy). Moreover, in the simple example that we showed, the two visits are invoked after each other at the root. In practice, for example in type checking languages with principal types, we actually invoke multiple visits on a subtree before moving on to the next subtree. This rules out depth-first visitors, and is also error-prone to write manually.

The example in Figure 1 can easily be made more complicated, for example by having menus that share submenus, and form an acyclic graph instead of a tree. With each of such complications, the above mentioned problems grow worse. As a sidenote, in this paper, we treat the AST as a fixed datastructure. For example, we do not consider adding menu entries on the fly. The ideas we propose can deal with the dynamic construction of proof trees [14], and we think that this is sufficient to deal with dynamic changes to the AST as well, but leave this topic as future work.

Below, we look for a way to generate code similar to the code above, but from a description that does not have the aforementioned problems.

2.2 Attribute grammars

Attribute grammars take care of the problems mentioned above related to visitors, but are not flexible enough to take side effect into account. We briefly consider why attribute grammars appear a promising solution, and why side effect is a problem. Before we show the example, we first give some background information on attribute grammars, and their encoding in JavaScript. As syntax, we take a mixture of UUAG's syntax [20], and ruler-front (which are closely related).

An attribute grammar is an extension of a context-free grammar, where nonterminals are annotated with attributes, and productions specify equations between attributes. The context-free grammar specifies the structure of the AST: each node of the AST is associated with a production. A node is also associated to the nonterminal of the left-hand side of the production, and each child of a node to the corresponding nonterminal in the right-hand side of the production.

For example, we can denote a production as well as the structure of a node in the AST using a data-type definition (explained below).

```
data Menus                          -- nonterminal Menus
  con Cons hd:Menu tl:Menus         -- production Cons, with two nonts
  con Nil                           -- production Nil, empty
```

This data-type declaration introduces a nonterminal *Menus* with two productions, representing a cons-list. The first production is named *Cons*, and corresponds in BNF to $Menus \rightarrow Menu\ Menus$. The two nonterminals *Menu* and *Menus* in the right-hand side (RHS) have explicitly been given the respective names *hd* and *tl*.

```
data Root  con Root  root : Menu                    -- node with a child named root
data Menu  con Menu  name  cs : Menus               -- node with a property name, and a child cs
type Menus : [Menu]                                 -- conceptually a cons-list, physically an array

var root = new Root_Root (                          -- the Menus are physically represented
  new Menu_Menu ("a", [                             -- as an array. However, conceptually
    new Menu_Menu ("b", [                           -- we define its attributes using the
      new Menu_Menu ("c", [])                       -- above cons-list representation.
    , new Menu_Menu ("d", [])])]));

attr Menu Menus inh depth finMax count              -- gathMax: width of submenu
                syn gathMax count                   -- two attributes with the name count

function align (root, anchor) {                     -- uses embedded attribute grammars
  datasem Root clause Root                          -- equations of production Root of nont Root
    root : depth    = 0                             -- initial depth
    root : count    = 0                             -- initial count
    root : finMax   = root : gathMax                -- choose gathered max as global max

  datasem Menu clause Menu                          -- production Menu of nonterm Menu
    cs : depth      = 1 + lhs : depth               -- increase depth for submenus
    cs : count      = 1 + lhs : count               -- increase count
    lhs : count     = cs : count                    -- provide the updated count to the parent

    loc : elem      = document.getElementById (loc : name)
    loc : offset    = lhs : depth * 20              -- indentation
    loc : width     = loc : offset + loc : elem.clientWidth
    lhs : gathMax   = Math.max (cs : gathMax, loc : width)
    cs : finMax     = lhs : finMax                  -- pass down final maximum

    loc : dummy     = (function () {                -- side-effectful statements
                        loc : elem.style.left   = (anchor.offsetLeft + loc : offset) + "px";
                        loc : elem.style.top    = (anchor.offsetTop + lhs : count * 30) + "px";
                        loc : elem.style.width  = (lhs : finMax − loc : offset) + "px";
                        loc : elem.style.height = 30 + "px";
                      }) ()                         -- directly call the anonymous function

  datasem Menus                                     -- equations of productions Cons and Nil
    clause Cons
      hd : depth    = lhs : depth                   -- pass depth downwards through the menus
      tl : depth    = lhs : depth

      hd : count    = lhs : count                   -- thread the count through the menus, in an
      tl : count    = hd : count                    -- in-order fashion. First to the head, then to
      lhs : count   = tl : count                    -- the tail, then back up to the parent.

      lhs : gathMax = Math.max (hd : gathMax, tl : gathMax)
      hd : finMax   = lhs : finMax                  -- pass global maximum downwards
      tl : finMax   = lhs : finMax

    clause Nil
      lhs : count   = lhs : count                   -- thread count through without changing it

      lhs : gathMax = 0                             -- initial maximum

  var inhs = new Inh_Root ();                       -- contains inh attrs of the root
  eval_Root (sem_Root, root, inhs);                 -- run the attribute evaluator
}
```

Fig. 2: Attribute grammar-based near-solution to menu alignment.

Terminals only have a name (shown later in Figure 2).

Furthermore, this data-type declaration introduces JavaScript constructor functions to construct ASTs. Each production is mapped to a constructor function that gets as parameter an object corresponding to the symbols in the RHS of the production. Each nonterminal is mapped to a constructor function that creates a base object that each of the objects corresponding to the productions inherits. Due to the inheritance, we can verify at the point of construction that the AST matches

the grammar.

```
function Menus () { }                -- nonterminal Menus: base class
function Menus_Cons (hd, tl) {       -- production Cons: subclass
  this.hd = hd; assert (hd instanceof Menu);
  this.tl = tl; assert (tl instanceof Menus);
}
Menus_Cons.prototype                 = new Menus ();
Menus_Cons.prototype.constructor = Menus_Cons;
function Menus_Nil () { }            -- production Nil: subclass
Menus_Nil.prototype                  = new Menus ();
Menus_Nil.prototype.constructor      = Menus_Nil;
```

Cons-lists occur often. As a shortcut, we alternatively write the following shorthand for the above instead.

```
type Menus : [Menu]
```

As an additional bonus, we can represent a list of menus as a Javascript array.

Evaluation of an attribute grammar runs an evaluation algorithm on each node, derived from the equations of its associated production, that decorates each node with attributes. We assume that attributes are physically represented as Javascript properties of the AST objects. Nodes are decorated with two types of attributes: inherited attributes are computed during evaluation of the parent of that node, and synthesized attributes are computed during evaluation of the node itself.

We declare the attributes of a nonterminal using an attribute declaration.

```
attr Menu inh depth      -- inherited attribute
          syn gathMax    -- synthesized attribute
```

These attribute names are mapped to object properties named *_inh_depth* and *_syn_gathMax*. At some point during attribute evaluation, given a participating *Menu* object m, the objects properties m *._inh_depth* and m *._syn_gathMax* will be defined. An inherited attribute may have the same name as a synthesized attribute: they are mapped to differently named properties. As an aside, nodes may define a number of local attributes, which can be seen as local variables.

To give a semantics to these attributes, we specify equations (rules) per production (explained below - full details of the nonterminal and its semantics in Figure 2).

```
datasem Menu    -- nonterminal Menu
  clause Menu   -- production Menu
    cs : depth     = 1 + lhs : depth                          -- rule
    loc : width    = 20 * lhs : width                         -- rule
    lhs : gathMax  = Math.max (loc : width, cs : gathMax)     -- rule
```

The left-hand side of an equation designates an inherited attribute, using the notation *childname* : *attrname*, which allows us to distinguish attribute names from properties. The names *loc* and *lhs* are special: *loc* indicates a local attribute, and *lhs* refers to a synthesized attribute of the current node. Thus, the attributes we need to define appear as left-hand side. For example, the above attribute designations are refer to the JavaScript properties *this.cs ._ inh_depth*, *this ._ loc_width*, and *this ._ syn_gathMax* respectively.

Similarly, the right-hand side consists of a JavaScript expression, with embedded attribute references. In this case, we may refer to the synthesized attributes of children, or with *lhs* to the inherited attributes of the current node. The terminals of a production are available as local attributes. In production *Menu*, there is a terminal called *name*, which is available as attribute *loc*:*name*. The translation of attribute references is similar as described above.

The last rule expands to the JavaScript statement:

$$this\ ._\ syn\_gathMax = Math.max\ (this\ ._\ loc\_width, this.cs\ ._\ syn\_gathMax);$$

Evaluation of an attribute grammar corresponds to traversing the AST one or more times, and executing rules, according to an evaluation strategy. In this paper, we restrict ourselves to the class of well-defined attribute grammars, whose attribute dependencies can be statically proved to be acyclic [11]. For those grammars, a traversal is possible that visits each subtree a bounded number of times. This corresponds precisely with typical uses of the visitor-design pattern.

Out of the semantic definitions for e.g. *Menu*, a function *sem_Menu* is generated containing the evaluation algorithm. Furthermore, to interface with the decorated tree from JavaScript code, a function *eval_Menu* is generated that takes the AST, the function *sem_Menu*, and an object containing values for the inherited attributes. It applies the semantic value, and returns an object with the synthesized attributes.

```
var inhs = new Inh_Menu ();
inhs.depth = 0;                                  -- provide inh attrs of root
syns = eval_Menu (sem_Menu, menu, inhs);         -- initiate evaluation
window.alert (syns.gathMax);                     -- access syn attrs of root
```

In Figure 2, we show an attribute grammar version of the example presented earlier. It is a non-solution, for reasons explained later, but exhibits various important properties. The keywords written in bold indicate a switch from JavaScript code to AG code, and layout determines the switch back.

The attribute grammar code starts with a number of data type definitions that describe the structure of the menu tree. We then define a number of attributes. In particular, the idea is that we gather a maximum *gathMax* (synthesized), and use its value at the root, to pass down the global maximum *finMax* (inherited). Moreover, we count the menus. The inherited attribute *count* specifies the count for the current menu, and the synthesized *count* is the count incremented with the total number of children.

We define the semantics for these attributes in the function *align*. Because *root* and *anchor* are its parameters, we also have access to these in the right-hand sides of rules.

To layout the HTML item, we need to execute a number of statements, and encode this as an expression. In JavaScript, this can be accomplished in a variety of ways. In the example, we choose to use a parameterless anonymous function.

In the semantic of *Menus*, rules are given to compute the attributes for lists of menus. These rules follow standard patterns: a topdown passing of *depth* and *finMax*, bottomup computation of gathMax, and an inorder threading of *count*. In the visitor-example, the fields in the visitor combined with side-effect took care of this behavior. With attribute grammars, we have to describe it explicitly. However, there are mechanisms to abstract from these patterns, in the form of copy rules [20], collection rules [13], or a generalization called default rules [14]. With such abstractions, the semantics of *Menus* can be written in a much conciser way (as we see later).

The AG code has several nice properties. The order of appearance of the rules is irrelevant. This allows the rules for e.g. *depth* and *count* to be written separately and merged automatically [20]. In the example, we give all the rules in one go to fit the page, however, for bigger projects the ability to write such rules separately is important to write coherent code.

Another nice property is the absence of invocations of visits (the *accept* calls in the visitor-example). The number of visits is totally implicit. From the dependencies between attributes in the rules, the attribute evaluator determines automatically that the attribute *root*:*gathMax* (in the semantics of *Root*) must be computed first in a visit, before it can be passed as *root*:*finMax*.

Finally, we check statically if there is an evaluation order of statements such that all attributes are defined before their value is accessed. The attribute declarations describe the attributes that must be defined, and those that are available. The rules describe what attributes must be available before computing an attribute, and an evaluation order is possible if the transitive closure of the dependencies is non-cyclic [11].

However, the above code has a number of problems, because the order of evaluation of rules is determined only by dependencies on attributes. In particular, the side-effect that rearranges the HTML items is not a dependency of any rule. Thus it is not clear when it is evaluated, if it is evaluated at all. Similarly, it is neither clear at what moment the widths of the HTML items are obtained. When there are other rules in play that have side effect that effects these widths, the interleaving of these side effects becomes even harder to predict. Finally, the root of the tree does not have any attributes defined, so there is actually no reason to expect any of the rules to be executed in the first place.

2.3 Ruler-front

We now present a solution using RULER-FRONT. The syntax of RULER-FRONT resembles the syntax of the AG in Figure 2, but is different. Before we jump into the

```
data Root  con Root  root:Menu                  -- node with a child named root
data Menu  con Menu name cs:Menus               -- node with a property name, and a child cs
type Menus:[Menu]                               -- conceptually a cons-list, physically an array

var root = new Root_Root (                      -- the Menus are physically represented
  new Menu_Menu ("a",[                          -- as an array. However, conceptually
    new Menu_Menu ("b",[                        -- we define its attributes using the
      new Menu_Menu ("c",[])                    -- above cons-list representation.
     ,new Menu_Menu ("d",[])])]));

itf Root                                        -- itf for nonterminal Root (root node)
  visit perform                                 -- one visit, named perform
    inh ast                                     -- menu-AST is inherited attribute

itf Menu Menus                                  -- itf for nonterminals Menus (menu nodes)
  visit gather                                  -- first visit: compute maximum
    inh ast depth                               -- needs AST and depth
    syn gathMax                                 -- computes maximum width of the menu
  visit layout                                  -- second visit: layout the HTML items
    inh finMax count                            -- needs global maximum width
    syn count                                   -- produces updated count

function align (root, anchor) {                 -- uses embedded attribute grammars
  datasem Root clause Root                      -- equations of production Root of nonterm Root
    root:depth   = 0                            -- initial depth
    root:count   = 0                            -- initial count
    root:finMax  = root:gathMax                 -- global max is the gathered max here
    invoke layout of root                       -- require that visit layout of root is invoked

  datasem Menu clause Menu                      -- equations scheduled to visits of Menu
    cs:depth     = 1 + lhs:depth                -- increase depth for submenus
    cs:count     = 1 + lhs:count                -- increase count
    lhs:count    = cs:count                     -- provide the updated count to the parent

    match loc:elem = document.getElementById (loc:name)
    loc:offset     = lhs:depth * 20             -- indentation
    loc:width      = loc:offset + loc:elem.clientWidth
    lhs:gathMax    = Math.max (cs:gathMax, loc:width)
    cs:finMax      = lhs:finMax                 -- pass down final maximum

    visit layout                                -- equations for visit layout and later
      match _ =    (function () {               -- side-effectful statements (wrapped as function)
                     loc:elem.style.left   = (anchor.offsetLeft + loc:offset) + "px";
                     loc:elem.style.top    = (anchor.offsetTop + lhs:count * 30) + "px";
                     loc:elem.style.width  = (lhs:finMax - loc:offset) + "px";
                     loc:elem.style.height = 30 + "px";
                   }) ()                        -- directly call the anonymous function

  datasem Menus                                 -- standard patterns for Menus
    default depth    = function (depths) {return depths [depths.length - 1]; }
    default finMax   = function (maxs)   {return maxs [maxs.length - 1]; }
    default gathMax  = function (maxs)   {return Math.max.apply (Math, maxs); }
    default count    = function (counts) {return counts [0]; }
    clause Cons                                 -- a clause must be given for each production,
    clause Nil                                  -- otherwise easy to forget one

  var inhs   = new Inh_Root_perform ();         -- contains inh attrs for the root
  inhs.ast   = root                             -- AST as inherited attribute
  eval_Root_perform (sem_Root, inhs);           -- run the attribute evaluator
}
```

Fig. 3: RULER-FRONT solution to menu alignment.

example, we first discuss some of the differences.

The central idea is to make visits to an AST node during attribute evaluation explicit. We then associate side effect with individual visits.

Interfaces. Instead of declaring attributes for a nonterminal, we declare an *interface* for a nonterminal. An interface declaration specifies the visits of a nonterminal,

```
function align (root, anchor) {                    -- uses embedded attribute grammars
  var sem_Root =                                   -- semantic function with itf Root
    sem ntRoot : Root                              -- equations for itf Root
      visit perform                                -- equations for the perform, the only visit
        clause Root                                -- production named Root
          child root : Menu = sem_Menu             -- introduce a child root of nonterm Menu
          root : ast        = lhs : ast            -- use lhs : ast as AST
          root : depth      = 0                    -- initial depth
          root : count      = 0                    -- initial count
          root : finMax     = root : gathMax       -- global max is the gathered max of here
          invoke layout of root                    -- demand invocation layout of root
  var sem_Menu =                                   -- semantic function with itf Menu
    sem ntMenu : Menu                              -- equations for itf Menu
      visit gather                                 -- equations for first visit
        clause Menu                                -- production named Menu
          child cs : Menus = sem_Menus             -- introduce a child cs of nonterm Menus
          cs.ast           = lhs : ast.cs          -- pass submenus as AST for cs
          cs : depth       = 1 + lhs : depth       -- increase depth for submenus
          match loc : elem = document.getElementById (loc.name)
          loc : offset     = lhs : depth * 20      -- indentation
          loc : width      = loc : offset + loc : elem.clientWidth
          lhs : gathMax    = Math.max (cs : gathMax, loc : width)
          cs : finMax      = lhs : finMax          -- pass down global maximum
          visit layout                             -- equations for visit layout
            clause Menu′                           -- subproduction named Menu′
              cs : count   = 1 + lhs : count       -- increase count
              lhs : count  = cs : count            -- provide the updated count to the parent
              match _      = (function () {        -- side-effectful statements
                    loc : elem.style.left   = (anchor.offsetLeft + loc : offset) + "px";
                    loc : elem.style.top    = (anchor.offsetTop + lhs : count * 30) + "px";
                    loc : elem.style.width  = (lhs : finMax − loc : offset) + "px";
                    loc : elem.style.height = 30 + "px";
                  }) ()                            -- directly call the anonymous function
  var sem_Menus =                                  -- semantic function, also itf Menu
    sem ntMenus : Menu                             -- equations for itf Menu
      visit gather                                 -- equations for visit gather
        default depth    = function (depths) {return depths [depths.length − 1]; }
        default finMax   = function (maxs)   {return maxs [maxs.length − 1]; }
        default gathMax  = function (maxs)   {return Math.max.apply (Math, maxs); }
        default count    = function (counts) {return counts [0]; }
        clause Cons                                -- production Cons
          match true = lhs : ast.length ⩾ 1        -- clause matches if array has an element
          child hd : Menu = sem_Menu               -- introduce child hd using sem_Menu
          hd.ast          = lhs : ast [0]          -- head of the array
          child tl : Menu = sem_Menus              -- introduce child tl using sem_Menus
          tl.ast          = lhs : ast.slice (1)    -- tail of the array
        clause Nil                                 -- production Nil (matches always)
  var inhs    = new Inh_Root_perform ();           -- contains inh attrs for the root
  inhs.ast    = root                               -- AST as inherited attribute
  eval_Root_perform (sem_Root, inhs);              -- run the attribute evaluator
}
```

Fig. 4: Desugared RULER-FRONT solution to menu alignment.

and attributes per visit. In the following example, we specify that the attributes of *Menu* are computed in two visits.

```
itf Menu                    -- interface for nonterminal Menu
  visit gather              -- declaration of first visit
```

```
  inh ast              -- inherited attr defined prior to visit
  syn gathMax          -- synthesized attr computed by visit
visit layout           -- declaration second visit
  inh finMax count     -- two inherited attributes
  syn count            -- synthesized attr computed by visit
```

The order of appearance of visit declarations dictates the order of visits to AST nodes with this interface. In order to visit a node, all previous visits must have occurred: the actual visits on a node must be a prefix of the declared visits. Values for inherited attributes must be provided prior to the visit. Values for synthesized attributes are only available after a visit has been performed.

In a conventional AG, the AST to traverse can be seen as hidden inherited attribute. In RULER-FRONT, the AST must actually be provided explicitly as inherited attribute *ast* in the first visit. Section 2.4 motivates this choice.

Scheduling. The rules of a semantics-block are automatically scheduled over visits using an as-late-as-possible strategy. If the rules are cyclicly defined, the scheduling is not possible, and a static error is reported. Visits to children are automatically inferred based on the attribute requirements of rules. However, since *Root* has no attributes, there is no need to invoke any visits of *root*. Therefore, we specify through an **invoke** rule that visit *layout* must be invoked, which requires through attribute dependencies that also visit *gather* must be invoked, and kickstarts the evaluation.

Scheduling constraints. Rules can be constrained to visits. With a visit-block, we constrain rules to that visit, or a later visit. The example below illustrates the various possibilities. An attribute definition prefixed with the keyword **match** is an exception. It is constrained to the visit it appears in, and is executed even if the attribute it defines is never needed. We explain its precise meaning later.

```
datasem Menu                    -- rules for nonterminal Menu
  clause Menu                   -- rules for production Menu
    cs : count = lhs : count + 1 -- scheduled in visit gather or later
    match loc : elem = ...      -- precisely in visit gather
    visit layout                -- rules for visit layout or later
      match _ = ...             -- precisely in visit layout
      lhs : count = cs : count  -- constrained to layout or later
```

With an underscore, we bind the value of the RHS of a rule to an anonymous attribute that we cannot refer to.

A visit-block also introduces a subscope. A local attribute defined in a visit-block is not available for a rule defined in a higher scope, even if that rule is scheduled to a subscope.

After all these preparations, we can finally present the RULER-CORE solution in Figure 3. In this example, we express that the side effect that queries the widths of the HTML items, is constrained to the first visit, and the side effect that changes the location and dimensions is constrained to the second.

For the *Menus*-nonterminal, we give default-rules for equality named attributes in its productions. If such an attribute does not have an explicit definition, these are implicitly defined by the default rule. The idea is that the default-rule provides a function that gets an area with all attribute values of the same name of previously visited children (or *lhs*). Formally, given a default-rule for attribute a, suppose that a child $k_i \in k_1, ..., k_n, lhs$ has an attribute $k.a$ (synthesized if $k = lhs$, inherited otherwise), but lacks an explicit definition for it. The default-rule gives an implicit definition, by invoking the RHS of the default-rule with an array defined as follows. For each child $c_j \in k_{i-1}, ..., k_1, lhs$ that has an attribute a (inherited if $k = lhs$, synthesized otherwise), in this order, the array has a value $c_j.a$. In particular, the first entry is the value of the closest child, and the last entry is that of *lhs* (if such attributes exist).

In the above example, we combined both side effect and attribute evaluation. We retain the advantages that AGs offer, such as the ease of adding attributes. Furthermore, the extension is orthogonal to various optimizations for attribute grammars, including incremental evaluation and multi-core parallel evaluation.

However, we require the programmer to manually assign attributes to visits, and constrain side-effectful rules to particular visits, which is not necessary for conventional attribute grammars. In practice, this is only a minimal amount of extra work that has as additional advantages that it makes attribute evaluation more predictable and thus easier to understand.

2.4 Desugared Ruler-Front

In Figure 4 (explained below), we give another way to write the same example in RULER-FRONT. Both Figure 4 and Figure 3 are valid RULER-FRONT programs. The former is, however, a desugared version of the latter. This desugared version only uses a subset of RULER-FRONT that we call RULER-CORE. It naturally generalizes over Higher-Order [23] and Conditional [2] Attribute Grammars. We use this example as preparation for RULER-CORE in the next section. To save space, we omitted the data-type declarations, interface declaration, and *root* variable, which are equal to those in the first half of Figure 3.

We present sem-blocks of the form **sem** *nonterm* : *Interface*, which introduces a nonterminal *nonterm*, with visits and attributes described by *Interface*. The productions are not defined by a data-type definition, but through clauses and rules per visit, as we explain below. Additionally, the code generated from a semantics-block is a constructor-function that produces an AST node described by *Interface*, which we can store in a variable, and may use in rules.

In Figure 4, we start with a definition of the semantics for the root. The interface *Root* declares one visit. We generalize over productions for a nonterminal by having clauses for each visit. Each clause provides an alternative way to compute the attribute values. We thus give clauses for the visit *perform*, in this case only one clause.

Clauses and visits may contain rules. Rules given for a visit are in scope of all clauses declared for that visit. Rules for a clause are only visible in that clause.

We also see another type of rule, called a child-rule, which introduces a child. For example, we introduce a child *root*, with interface *Menu*, and the semantics defined by the JavaScript value *sem_Menu*.

The left-hand sides of an evaluation-rule may be a pattern. This is either an attribute reference, an underscore or a constant. Evaluation of such a rule fails when its execution throws an exception, or the left-hand side is a value that is not equal to the value computed for the right-hand side.

During attribute evaluation, the clauses of a visit are tried at runtime in the order of appearance. The next clause is tried when either a match-rule fails, or when there is no succeeding clause for a visit to a child. Failure of any other form of rule simply aborts the entire evaluation. This way, the match-rules allow us to distinguish clauses *Cons* and *Nil* of *ntMenus* by matching on the length of the list.

Missing visits are implicitly defined with a single empty clause. A visit without clauses implicitly has a single clause. Therefore, we neither have to specify the visit *layout* nor clauses for it in the semantics of *ntMenus*. Also, due to the automatic ordering of rules, many of the rules defined in visit *layout* of *ntMenu*, could also be defined one level higher, in visit *gather*.

Note that this representation is more general than conventional attribute grammars, and that an attribute grammar can easily be mapped to this representation, as shown by the difference between Figure 3 and Figure 4.

$e ::= \mathcal{J}\ [\overline{b}]$ -- embedded RULER-FRONT blocks b in $\mathcal{J}$
$b ::= i \mid s \mid o$ -- RULER-FRONT blocks

$i ::= \textbf{itf}\ I\ \overline{v}$ -- interface decl, with visits v
$v ::= \textbf{visit}\ x\ \textbf{inh}\ \overline{x_1}\ \textbf{syn}\ \overline{x_2}$ -- visit decl, with atributes x_1 and x_2

$s ::= \textbf{sem}\ x : I\ t$ -- semantics expr, defines nonterm x
$t ::= \textbf{visit}\ x\ \overline{r}\ \overline{c}$ -- visit def, with common rules r
$\mid\ \epsilon$ -- no visit (serves as terminator)
$c ::= \textbf{clause}\ x\ \overline{r}\ t$ -- clause definition, with next visit t

$r ::= p = e$ -- assert-rule, evaluates e, bind to pattern p
$\mid\ \textbf{match}\ p = e$ -- match-rule, backtracking variant
$\mid\ \textbf{invoke}\ x\ \textbf{of}\ c$ -- invoke-rule, invokes visit x on c
$\mid\ \textbf{child}\ c : I = e$ -- child-rule, introduces a child c of itf I

$o ::= c : x$ -- attribute reference in some embedded code

$p ::= c : x$ -- attribute reference in pattern
$\mid\ \_$ -- underscore
$\mid\ k$ -- constant

$x, c\ I, p, e$ -- identifiers, child identifiers, patterns, expressions respectively

$\Gamma, \Sigma ::= \epsilon$ -- attr+child environment (used in semantics)
$\mid\ \Gamma, \circ$ -- new scope
$\mid\ \Gamma, \textbf{inh}\ c : x$ -- inh attr $c : x$
$\mid\ \Gamma, \textbf{syn}\ c : x$ -- syn attr $c : x$
$\mid\ \Gamma, c : I\ \overline{v}$ -- child c with available visits $\overline{v}$

$\Phi ::= \epsilon$ -- interface environment (used in semantics)
$\mid\ \Phi, I\ \overline{v}$ -- itf I with visit decls v

Fig. 5: Syntax of RULER-CORE

3 Static Semantics of ruler-core

In this section, we introduce RULER-CORE, a small subset of RULER-FRONT, but sufficiently rich to serve as intermediate language for RULER-FRONT. Figure 5 lists the syntax of RULER-CORE. A RULER-FRONT program e is a *JavaScript* program J, with embedded RULER-CORE blocks b. A block b is either an interface declaration, semantics-block, or attribute reference. We explain the individual forms of syntax in more detail below.

There are some essential differences in contrast to RULER-FRONT that we gradually introduced by example in the previous section. The order of appearance of rules the evaluation order, and each invocation of a visit must explicitly be stated through an invoke rule. Special syntax for data-types is not part of RULER-CORE. Through clauses and (match) rules, we have a general mechanism to inspect and perform case distinction on arbitrary JAVASCRIPT datastructures.

We make no assumptions about the syntax of J. The embedded blocks may occur anywhere in a JAVASCRIPT program. It is up to the programmer to ensure that semantic-blocks and attribute references occur at expression-positions, and that interface-declarations occur at statement positions. Neither do we make any assumptions about scopes of J; instead, we assume that all embedded blocks are in the same scope.

```
itf S visit v1 inh l syn ∅          -- decompose array l down
    visit v2 inh ∅ syn s            -- compute sum s up
var sumArr = sem sum:S
  visit v1 ∅                        -- first visit
    clause sumNil                   -- when list is empty
      match 0 = lhs:l.length        -- match empty l
      visit v2 ∅                    -- second visit
        clause sumNil2              -- single clause
          lhs:s = 0                 -- empty list, zero sum
          ()                        -- no next visit
    clause sumCons                  -- when list non-empty
      loc:x  = lhs:l [0]            -- head of the list
      loc:xs = lhs:l.slice (1)      -- tail of the list
      child tl:S = sumArr           -- recursive call
      tl:l = loc:xs                 -- l param of call
      invoke v1 of tl               -- invoke on child
      visit v2 ∅                    -- second visit
        clause sumCons2             -- single clause
          invoke v2 of tl           -- invoke on child
          lhs:s = loc:x + tl:s      -- sum of head and tail
          ()                        -- no next visit
```

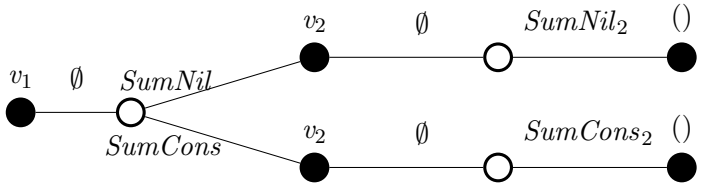
Fig. 6: Example of RULER-CORE syntax: summing an array of integers.

Figure 6 shows an example RULER-CORE program to sum an array of integers in two visits. The first visit has two clauses: a clause *sumNil* when the array is empty, and *sumCons* when there is at least one element. In the second visit, we compute the actual sum, depending on the clause chosen in the first visit.

A semantics-block introduces a visitor-object with an interface I. The interface

dictates what visits can be made to the object, and what the inputs (inherited attributes) and outputs are (synthesized attributes).

The outputs for a visit are produced by executing rules. We write these rules down in a tree of clauses and visits, as illustrated by the indentation in Figure 6 and the state diagram:



The black nodes represent the state of the AST-node prior to a visit, and the white nodes indicate a branch point. Upon creation, an AST node is in the state represented by the root node. With each edge are alternately associated the rules of a visit or of a clause. With each visit, an AST node tries to switch state to a next black node by executing the rules on the path to such a node. Execution of all of the rules must succeed. At a branch-point, rules on edges of clauses are tried in order of appearance. Results produced by executing rules are in scope of rules further along the path.

There are four types of rules.

- **match** $p = e$ -- match-rule
  match $loc : x = 3$ -- example that succeeds
  match $true \ = false$ -- example that fails

The pattern p must match the value of the right hand side. If the evaluation of e results in an exception, or the match fails, a backtrack is made to the next clause. If p represents an attribute, the attribute gets defined.

- $p = e$ -- assert-rule (not prefixed with a keyword)

Similar to the above, except that the match is expected to succeed. If not, the evaluation itself aborts with an exception.

- **child** $c : I = e$ -- child-rule
  child $root : Menu = ntMenu$ -- example that introduces a *Menu* child

Evaluation of the rule above creates a child c, visitable according to the interface I, and created by executing the constructor function e.

- **invoke** x **of** c -- invoke rule

Executes visit x of child c. The inherited attributes of x must be defined, and all prior visits to c must have been performed. The invocation may fail if no clause matches. In that case, it causes the current AST node to backtrack to the next clause. If successful, the synthesized attributes of x become available.

$$[\,]\ ;\ \Gamma \vdash \epsilon \ \textsc{end} \qquad \frac{\Gamma_0 \cup \mathit{avail}\ (\mathbf{visit}\ x\ \overline{r}\ \overline{c})\ ;\ \Gamma_0 \vdash \overline{r}\ :\ \Gamma_1 \quad \overline{v}\ ;\ \overline{s}\ ;\ \Gamma_1 \cup \{\mathbf{inh}\ \mathit{lhs}\!:\!a \mid a \in \overline{i}\} \vdash c_i}{\mathbf{visit}\ x\ \mathbf{inh}\ \overline{i}\ \mathbf{syn}\ \overline{s}, \overline{v}\ ;\ \Gamma_0 \vdash \mathbf{visit}\ x\ \overline{r}\ \overline{c}}\ \textsc{visit}$$

$$\frac{x\ \mathit{unique} \qquad \Gamma_0 \cup \mathit{avail}\ (\mathbf{clause}\ x\ \overline{r}\ \overline{c})\ ;\ \Gamma_0 \vdash \overline{r}\ :\ \Gamma_1 \quad \overline{v}\ ;\ \Gamma_1 \vdash t \qquad \{(\mathbf{syn}\ \mathit{lhs}\!:\!a) \mid a \in \overline{s}\} \subseteq \Gamma_1}{\overline{v}\ ;\ \overline{s}\ ;\ \Gamma_0 \vdash \mathbf{clause}\ x\ \overline{r}\ t}\ \textsc{clause}$$

$$\frac{\Sigma\ ;\ \Gamma_0 \vdash p\ :\ \Gamma_1 \qquad \Gamma_0 \vdash e}{\Sigma\ ;\ \Gamma_0 \vdash p = e\ :\ \Gamma_1}\ \textsc{assert} \qquad \frac{\Sigma\ ;\ \Gamma_0 \vdash p\ :\ \Gamma_1 \qquad \Gamma_0 \vdash e}{\Sigma\ ;\ \Gamma_0 \vdash p = e\ :\ \Gamma_1}\ \textsc{match}$$

$$\frac{\begin{array}{c}\Phi\ (I_c) = \overline{v} \qquad \mathbf{visit}\ x\ \mathbf{inh}\ \overline{i}\ \mathbf{syn}\ \overline{s} \in \overline{v} \\ c\!:\!I_c\ \overline{w} \in \Gamma_0 \qquad \mathit{next}\ \overline{w}\ \overline{v} = x \qquad \{\mathbf{inh}\ c\!:\!a \mid a \in \overline{i}\} \subseteq \Gamma_0 \\ \Gamma_1 = \Gamma_0 \cup \{\mathbf{syn}\ c\!:\!a \mid a \in \overline{s}\} \cup \{\,c\!:\!I_c\ (\overline{w}, \mathbf{visit}\ x\ \mathbf{inh}\ \overline{i}\ \mathbf{syn}\ \overline{s})\}\end{array}}{\Sigma\ ;\ \Gamma_0 \vdash \mathbf{invoke}\ x\ \mathbf{of}\ c\ :\ \Gamma_1}\ \textsc{invoke}$$

$$\frac{\Gamma_0 \vdash e \qquad \Gamma_1 = \Gamma_0 \cup \{\,c\!:\!I\ \emptyset\}}{\Sigma\ ;\ \Gamma_0 \vdash \mathbf{child}\ c\!:\!I = e\ :\ \Gamma_1}\ \textsc{child} \qquad \frac{\mathbf{inh}\ \mathit{lhs}\!:\!a \in \Gamma}{\Gamma \vdash \mathit{lhs}\!:\!a}\ \textsc{occ.lhs}$$

$$\frac{\mathbf{syn}\ c\!:\!a \in \Gamma}{\Gamma \vdash c\!:\!a}\ \textsc{occ.child} \qquad \frac{\mathbf{syn}\ \mathit{lhs}\!:\!a \in \Sigma}{\Sigma\ ;\ \Gamma_0 \vdash \mathit{lhs}\!:\!a\ :\ \Gamma_0, \mathbf{syn}\ \mathit{lhs}\!:\!a}\ \textsc{pat.lhs}$$

$$\Sigma\ ;\ \Gamma_0 \vdash \mathit{loc}\!:\!a\ :\ \Gamma_0, \mathbf{syn}\ \mathit{loc}\!:\!a\ \textsc{pat.loc} \qquad \frac{\mathbf{inh}\ c\!:\!a \in \Sigma}{\Sigma\ ;\ \Gamma_0 \vdash c\!:\!a\ :\ \Gamma_0, \mathbf{inh}\ c\!:\!a}\ \textsc{pat.child}$$

$$\Sigma\ ;\ \Gamma \vdash k\ :\ \Gamma\ \textsc{const} \qquad \Sigma\ ;\ \Gamma \vdash \_\ :\ \Gamma\ \textsc{any}$$

$$\begin{array}{ll}
\mathit{avail}\ (\mathbf{visit}\ x\ \overline{r}\ \overline{c}) & = \mathit{avail}_{\cup}\ (\overline{r}) \cup \mathit{avail}_{\cap}\ (\overline{c}) \\
& \quad \cup \{\mathbf{syn}\ \mathit{lhs}\!:\!b \mid \mathbf{visit}\ x\ \mathbf{inh}\ \overline{a}\ \mathbf{syn}\ \overline{b} \in \Phi\ (I_x)\} \\
\mathit{avail}\ (\mathbf{clause}\ x\ \overline{r}\ t) & = \mathit{avail}_{\cup}\ (\overline{r}) \cup \mathit{avail}\ (t) \\
\mathit{avail}\ (p = e) & = \emptyset \\
\mathit{avail}\ (\mathbf{match}\ p = e) & = \emptyset \\
\mathit{avail}\ (\mathbf{invoke}\ x\ \mathbf{of}\ c) & = \{\mathbf{inh}\ c\!:\!a \mid a \in \overline{a}, \mathbf{visit}\ x\ \mathbf{inh}\ \overline{a}\ \mathbf{syn}\ \overline{b} \in \Phi\ (I_c)\} \\
\mathit{avail}\ (\mathbf{child}\ c\!:\!I = e) & = \{\,c\!:\!I\ (\Phi\ I)\}
\end{array}$$

Fig. 7: Static semantics of RULER-CORE

Figure 7 shows a static semantics for RULER-CORE. A RULER-CORE program that satisfies these conditions never crashes due to an undefined attribute, invalid rule order, or forgotten invocation to a child. Dynamic or static type checking we leave as responsibility of the host language.

We briefly consider some aspect of these rules. Two environments play an important role: Γ represents the children and attributes defined so far (to test for

missing and duplicated definitions), and Σ the attributes that are allowed to be defined (to test for definitions of unknown attributes). As additional constraint on environments, we consider it a static error when there is a duplicate attribute in the environment within two scope markers.

Visits must be specified in the proper order, and none may be omitted. The relation for visits t gets a sequence of pending visits $\overline{v}$ as declared in the interface. In rule VISIT, we verify that the name of the visit matches the expected visit in the head of $\overline{v}$. The next visit must match the head of the tail of this list, until in the end $\overline{v}$ is empty. We also add the inherited attributes of the visits to the environment.

The function *avail* defines which attributes may be defined. Higher-up in the visit-clauses-tree, we may only define those attributes that are common to all lower clauses. In rules PAT.LHS and PAT.CHILD, we verify that we are indeed defining an attribute belonging to a certain child.

In rule INVOKE, we verify that x is indeed the next visit in the expected sequence of visits $\overline{v}$, given the previous invocations $\overline{w}$. We furthermore verify that the inherited attributes for the visit of c are defined, and add the synthesized attributes to the environment.

4 Translation of ruler-core to JavaScript

In this section, we describe how to translate RULER-CORE programs to JAVASCRIPT. We translate each semantics-block to a coroutine, implemented as one-shot continuations. Each call to the coroutine corresponds with a visit. The parameters of the coroutine are the inherited attributes of the visit. The result of the call is an object containing values for the synthesized attributes, and the continuation to call for the visit.

As an example, we show in Figure 8 the translation of the example in the previous section. To deal with backtracking, we use the exception mechanism, and throw an exception to switch to the next clause. Note that this does not rollback any side effect that the partial execution of the rules may have caused. To be able to do so, we can run the rules in a software transaction [9], for which many programming languages have tool support nowadays. Alternatively, when the side effect matters, the programmer can schedule it to an earlier or later visit, such that it is not influenced by backtracking.

To deal with continuations, we use closures. The function to be used for the next visit, we build in the previous visit. This function has access to all the results computed in the previous visit. Furthermore, we store values for attributes in local variables. Those values that are not needed anymore, are automatically cleaned up by the garbage collector.

Figure 9 shows the general tranlation scheme, and naming scheme for attributes. In particular, for each visit, we generate a closure that takes values for inherited attributes as parameter. Clauses are dealt with through exception handling. When a clause successfully executed all statements, it returns an object containing values for synthesized attributes, as well as the continuation function for the next visit.

```
var sumArr = function () {                  -- semantic function
  function nt_sum (_inps) {                 -- visit v1
    var lhsIl = _inps.l;                    -- extract lhs:l
    try {                                   -- try clause sumNil
      if (lhsIl.length != 0) throw eEval;   -- if lhs:l is empty

      var _res = new Object ();             -- produce results of v1
      _res._next = function (_inps) {       -- cont. for visit v2
          var lhsSs  = 0;                   -- lhs:s rule

          var _res   = new Object ();       -- produce results of v2
          _res._next = null;                -- no next visit
          _res.s     = lhsSs;               -- store lhs:s
          return _res;                      -- return result of v2
        };
      return _res;                          -- return result of v1
    } catch (err) {                         -- try clause sumCons
        var locLx  = lhsIl [0];             -- loc:x rule
        var locLxs = lhsIl.slice (1);       -- loc:xs rule
        var vis_tl = sumArr ();             -- creation of child tl
        tlIl = locLxs;                      -- tl:l rule

        var _args  = new Object ();         -- inputs for v1 of tl
        _args.l    = tlIl;                  -- store tl:l
        var _res   = vis_tl (_args);        -- invoke v1 of tl
        var vis_tl = _res._next;            -- extract results

        var _res = new Object ();           -- produce results of v1
        _res._next = function (_inps) {     -- cont. for visit v2
          var _args  = new Object ();       -- inputs for v2 of tl
          var _res   = vis_tl (_args);      -- invoke v2 of tl
          var tlSs   = _res.s;              -- extract tl:s result

          var lhsSs  = locLx + tlSs;        -- compute lhs:s

          var _res   = new Object ();       -- produce results of v1
          _res._next = null;                -- no next visit
          _res.s     = lhsSs;               -- store lhs:s
          return _res;                      -- return result of v2
          };
      return _res;                          -- return result of v1
    }};return nt_sum; };                    -- return visitor function
```

Fig. 8: Example translation

The above translation is relatively straightforward. In practice, the selection of a clause is functionally dependent on the value of an inherited attribute, or a local attribute computed in a previous visit. In those cases, the selection of clauses can be implemented more efficiently using conventional branching mechanisms.

We verified that the above implementation runs in time linear to the size of the tree, when we use version of the *slice* operation that does not make a copy of the array. With a throughput of about hundred array elements per microsecond, and about a thousand per microsecond with the exception handling replaced by conventional branching, this is still about one or two orders of magnitude slower than using a hand-written loop. In our experience, however, performance is rarely an issue. In general, the asymptotic complexity of the traversal is linear in the size of the tree, and the actual time taken by traversing the trees is insignificant compared to the work performed by the right-hand sides of the rules in a real application.

$$
\begin{array}{ll}
[\![\textbf{sem}\ x : I\ t]\!] & \rightsquigarrow \textbf{function}\ ()\ \{\textbf{var}\ [\![nt\ x]\!] = [\![t]\!]_I; \textbf{return}\ [\![nt\ x]\!];\} \\
[\![c : x]\!] & \rightsquigarrow [\![inp\ c\ x]\!] \\
[\![()]\!]_I & \rightsquigarrow null \\
[\![\textbf{visit}\ x\ \overline{r}\ \overline{c}]\!]_I & \rightsquigarrow \textbf{function}\ (\_inps)\ \{ \\
 & \quad [\![inp\ lhs\ (inhs\ I\ x)]\!] = \_inps.[\![inhs\ I\ x]\!]; \\
 & \quad [\![\overline{r}]\!]; [\![\overline{c}]\!]_{I,syns\ I\ x};\} \\
[\![[\,]]\!]_{I,\overline{s}} & \rightsquigarrow \textbf{throw}\ eEval; \\
[\![c : \overline{c}]\!]_{I,\overline{s}} & \rightsquigarrow \textbf{try}\ \{[\![c]\!]_{I,\overline{s}};\} \\
 & \quad \textbf{catch}\ (err)\ \{ \\
 & \quad\quad \textbf{if}\ (err == eEval)\ \{[\![\overline{c}]\!]_{I,\overline{s}};\} \\
 & \quad\quad \textbf{else throw}\ err;\} \\
[\![\textbf{clause}\ x\ \overline{r}\ t]\!]_{I,\overline{s}} & \rightsquigarrow [\![\overline{r}]\!]; \\
 & \quad \textbf{var}\ \_outs = new\ Object\ (); \\
 & \quad \_outs.\_next = [\![t]\!]_I; \\
 & \quad \_outs.\overline{s} = [\![out\ lhs\ \overline{s}]\!]; \\
 & \quad \textbf{return}\ \_outs; \\
[\![p = e]\!] & \rightsquigarrow \textbf{var}\ \_res; \textbf{try}\ \{\_res = [\![e]\!];\}\ \textbf{catch}\ (err)\ \{ \\
 & \quad\quad \textbf{if}\ (err == eEval)\ \textbf{throw}\ eAbort; \textbf{else throw}\ err;\} \\
 & \quad [\![p]\!]_{eAbort} \\
[\![\textbf{match}\ p = e]\!] & \rightsquigarrow \textbf{var}\ \_res = [\![e]\!]; [\![p]\!]_{eEval}; \\
[\![\textbf{child}\ c : I = e]\!] & \rightsquigarrow \textbf{var}\ [\![vis\ c]\!] = ([\![e]\!])\ (); \\
[\![\textbf{invoke}\ x\ \textbf{of}\ c]\!] & \rightsquigarrow \textbf{var}\ \_args = new\ Object\ (); \\
 & \quad \_args.[\![inhs\ I_c\ x]\!] = [\![out\ c\ (inhs\ I_c\ x)]\!]; \\
 & \quad \textbf{var}\ \_res = [\![vis\ c]\!]\ (\_args); \\
 & \quad \textbf{var}\ [\![inp\ c\ (syns\ I_c\ x)]\!] = \_res.[\![syns\ I_c\ x]\!]; \\
 & \quad \textbf{var}\ [\![vis\ c]\!] = \_res.\_next; \\
[\![c : a]\!]_e & \rightsquigarrow \textbf{var}\ [\![out\ c\ a]\!] = \_res; \\
[\![\_]\!]_e & \rightsquigarrow ; \\
[\![k]\!]_e & \rightsquigarrow \textbf{if}\ (\_res\ !\ = k)\ \textbf{throw}\ e;
\end{array}
$$

$$
\begin{array}{ll}
out\ \texttt{"loc"}\ x = \texttt{"locL"}\ x & inp\ \texttt{"loc"}\ x = \texttt{"locI"}\ x \\
out\ \texttt{"lhs"}\ x = \texttt{"lhsS"}\ x & inp\ \texttt{"lhs"}\ x = \texttt{"lhsI"}\ x \\
out\ c\quad x = c\ \texttt{"I"}\ x & inp\ c\quad x = c\ \texttt{"S"}\ x \\
vis\ c = \texttt{"vis\_"}\ c & nt\ x = \texttt{"nt\_"}\ x \\
syns\ I\ x \quad inhs\ I\ x & \text{-- respectively, inh and syn attrs of } x \text{ of } I
\end{array}
$$

Fig. 9: Denotational semantics of RULER-CORE

5 Related Work

Related to this paper are various visitor-like approaches and attribute grammar techniques.

The purpose of the Visitor design pattern [7] is to decouple traversal operations from the specification of the tree to be traversed, in order to make it easier to add new operations without changing the existing specification of the tree. This allows us to write a multi-visit traversal using a separate visitor per traversal.

In Section 2.1, we discussed advantages and disadvantages of modeling traversals with this pattern. In particular, side effect is permitted, and used to store results for use in later visits. The side effect makes it hard to predict if results needed in a next visit are actually stored by a first visit. This is a fundamental problem of visitors. Oliveira, et al. [16], for example, show many enhancements with respect to the type safety of visitors, but do not address the transfer of results between visits.

Attribute grammars [11,12] were considered to be a promising implementation

for compiler construction, but several success stories aside, did not meet these expectations [24]. The bets may be turning again.

Recently, many Attribute Grammar systems arose for mainstream languages, such as Silver [25] and JastAdd [5] for Java, and UUAG [20] for Haskell. In contrast to the work in this paper, these systems strictly discourage or even forbit the use of side effect. The design of RULER-CORE is inspired by the language of execution plans of UUAG. In certain languages it is possible to implement AGs via meta-programming facilities, which obliviates the need of a preprocessor. Viera, et al. [22] show how to implement AGs into Haskell through type level programming. The ideas presented in this paper are orthogonal to such approaches, although the necessary dependency analysis may be difficult to express depending on the expressiveness of the meta language.

Several attribute grammar techniques are important to our work. Kastens [10] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, allowing severe optimizations.

6 Conclusion

We introduced the language RULER-FRONT, an extension of Attribute Grammars that makes visits to nonterminals explicit. As a consequence, it is possible to use side effects in rules. It combines the freedom of visitors as described by the Visitor Design Pattern with the convenience of programming with attributes, as shown in Section 2.

Moreover, we presented RULER-CORE, a subset of RULER-FRONT, which serves as a small core language for visitor-based Attribute Grammars. In RULER-CORE, the lifetime of attributes is explicit, as well as the evaluation order of rules and visits to children. A RULER-CORE program has a straightforward translation to many languages. In Section 4, we showed a translation to JavaScript .

There are many directions for future work. The parallel evaluation of Attribute Grammars received a lot of interest in the past, but during a time that multi-core processors were not commonly available. The small RULER-CORE language is suitable for experimentation with different evaluation strategies.

Another direction of research is to allow destructive updates on attributed trees. For example, to support event-handling traversals over data structures that are dynamically changed based on user input or external events. In RULER-FRONT, the visits performed on an attributed tree explicitly specify which attributes are defined. When we apply a destructive update to the tree, we thus know precisely what information is based upon the previous structure of the tree, which is beneficial when reasoning about mutations to the tree. Incremental evaluation of Attribute Grammars received attention in the past, and may be used to efficiently recompute attributes after an AST change.

More fundamentally, the idea of this paper is to deal with the scheduling of rules in the presence of side effect. This is not possible with conventional attribute gram-

mars, because the effects are not visible in attribute dependencies. In the Haskell version of RULER-FRONT, the left-hand side of a rule can be a match against a data constructor. If this data constructor is a GADT, the match brings type assumptions in scope, to be used to coerce types in rules that follow. Similarly to side effect, these type assumptions are implicit. However, with RULER-FRONT, we can explicitly schedule rules to be after such a match. This allows us to combine GADT features with Attribute Grammars. This may be sufficient to target dependently-typed programming languages, and a direction towards verified compilers using AGs.

Acknowledgement

This work was supported by Microsoft Research through its European PhD Scholarship Programme.

References

[1] Bieniusa, A. and A. Middelkoop, *JTransactifier: transactification of Java programs through annotations*, http://proglang.informatik.uni-freiburg.de/projects/dstm/ (2009).

[2] Boyland, J., *Conditional Attribute Grammars*, ACM TPLS **18** (1996), pp. 73–108.

[3] Bruneton, E., *ASM 3.0, a Java bytecode engineering library*, http://download.forge.objectweb.org/asm/asm-guide.pdf (2007).

[4] Dijkstra, A., J. Fokker and S. D. Swierstra, *The architecture of the Utrecht Haskell compiler*, in: *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell* (2009), pp. 93–104.

[5] Ekman, T. and G. Hedin, *The JastAdd Extensible Java Compiler*, in: R. P. Gabriel, D. F. Bacon, C. V. Lopes and G. L. S. Jr., editors, *OOPSLA* (2007), pp. 1–18.

[6] Gagnon, E. M. and L. J. Hendren, *SableCC, an Object-Oriented Compiler Framework*, in: *TOOLS (26)* (1998), pp. 140–154.

[7] Gamma, E., R. Helm, R. E. Johnson and J. M. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, in: O. Nierstrasz, editor, *ECOOP*, Lecture Notes in Computer Science **707** (1993), pp. 406–431.

[8] Heeren, B., D. Leijen and A. van IJzendoorn, *Helium, for Learning Haskell*, in: *ACM SIGPLAN Haskell Workshop (HW'03)* (2003), pp. 62 – 71.

[9] Heidegger, P., A. Bieniusa and P. Thiemann, *DOM Transactions for Testing JavaScript*, in: *TAICPART*, 2010, p. (to appear).

[10] Kastens, U., *Ordered Attributed Grammars*, Acta Inf. **13** (1980), pp. 229–256.

[11] Knuth, D. E., *Semantics of Context-Free Languages*, Math. Sys. Theory **2** (1968), pp. 127–145.

[12] Knuth, D. E., *The Genesis of Attribute Grammars*, in: *WAGA*, 1990, pp. 1–12.

[13] Magnusson, E., T. Ekman and G. Hedin, *Extending Attribute Grammars with Collection Attributes–Evaluation and Applications*, SCAM07 **0** (2007), pp. 69–80.

[14] Middelkoop, A., A. Dijkstra and S. D. Swierstra, *Iterative Type Inference with Attribute Grammars*, in: *Proceedings of the International Conference on Generative Programming and Component Engineering*, 2010.

[15] Middelkoop, A., A. Dijkstra and S. D. Swierstra, *Visitor-based Attribute Grammars with Side Effect (Extended Version)*, http://people.cs.uu.nl/ariem/wgt10-journal.pdf (2010).

[16] Oliveira, B. C. D. S., M. Wang and J. Gibbons, *The visitor pattern as a reusable, generic, type-safe component*, in: G. E. Harris, editor, *OOPSLA* (2008), pp. 439–456.

[17] Palsberg, J. and C. B. Jay, *The Essence of the Visitor Pattern*, in: *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference* (1998), pp. 9–15.

[18] Roberts, E., *An overview of MiniJava*, in: H. M. Walker, R. A. McCauley, J. L. Gersting and I. Russell, editors, *SIGCSE* (2001), pp. 1–5.

[19] Schrage, M. M. and J. T. Jeuring, *Proxima - A presentation-oriented editor for structured documents* (2004).

[20] Universiteit Utrecht, *Universiteit Utrecht Attribute Grammar System*, http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem.

[21] Universiteit Utrecht, *Mini Projects Compiler Construction*, http://www.cs.uu.nl/wiki/bin/view/Cco/MiniProjects (2010).

[22] Viera, M., S. D. Swierstra and W. Swierstra, *Attribute grammars fly first-class: how to do aspect oriented programming in Haskell*, in: G. Hutton and A. P. Tolmach, editors, *ICFP* (2009), pp. 245–256.

[23] Vogt, H., S. D. Swierstra and M. F. Kuiper, *Higher-Order Attribute Grammars*, in: *PLDI*, 1989.

[24] Waite, W. M., *Use of Attribute Grammars in Compiler Construction*, in: P. Deransart and M. Jourdan, editors, *WAGA*, Lecture Notes in Computer Science **461** (1990), pp. 255–265.

[25] Wyk, E. V., D. Bodin, J. Gao and L. Krishnan, *Silver: an Extensible Attribute Grammar System*, Electr. Notes Theor. Comput. Sci. **203** (2008), pp. 103–116.