Dynamic Attribute Grammars: a Declarative Functional Language

Martin Jourdan, Didier Parigot, Gilles Roussel, Étienne Duris

Avant-projet OSCAR — INRIA
B.P. 105
78153 Rocquencourt Cedex (France)
http://www-rocq.inria.fr/oscar/
mailto:First.Last@inria.fr

Attribute Grammars

- Introduced by Knuth in 1968
- Specification or programming paradigm:
  * Syntax-directed, structured, declarative
  * Well suited for “large” language processing applications
- 30 years of research:
  * efficient and versatile implementation techniques
  * accurate global analysis techniques
  * language extensions (modularity, genericity, etc.)
  * high-quality systems (Fnc-2, Linguist, Eli/LIGA)

In spite of that…

- Attribute Grammars are hardly used in commercial applications. Reasons:
  * Systems are not that good (???)
  * The AG paradigm is not expressive enough:
    - a physical tree is absolutely needed
    - the computation scheme is strongly tied to this tree
  * Extensions bringing more expressiveness are limited in scope or hard to use:
    - Circular AGs
    - Multi-Attributed Grammars
    - Higher-Order AGs
    - Door AGs
    - Conditional AGs

Our contribution: Dynamic Attribute Grammars

- The grammar describes the control flow rather than a physical object, and attribute values (data flow) can influence this flow of control
- Language extensions:
  * Grammar couples
    - Association of recursion schemes to (possibly empty) concrete trees
  * Semantic rule blocks
    - Conditions over attribute values allow to choose between several (dynamic) productions and associated semantic rules
- Implementation of DAGs uses the same techniques as traditional AGs
  * efficiency and versatility
  * accurate global analysis and optimizations

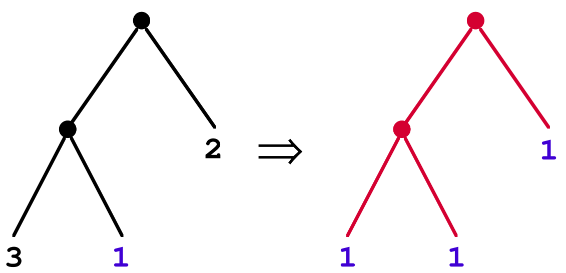
Benefits of Dynamic Attribute Grammars

- **A new language**:
  - as expressive as traditional first-order languages
  - retains the distinctive declarative character of AGs
  - efficient implementation
- **Cross-fertilization**:
  - establish links between some program in AG and the same program in other paradigms (functional, logic, etc.)
  - using these links, compare the implementations and analyses of these programs
  - take what is good in one paradigm and adapt it to the others

Outline of the talk

- Examples and comparisons with ML
- Visit-sequence-based implementation
- Examples of cross-fertilization

A Classical Example: Bird




```
type TREE = fork of TREE * TREE
   | tip of int;;
```


Bird as a Single Function

(Lazy ML)

```
let bird t = t1
  where rec (t1, n) = repmin t n
  where rec repmin t n =
    match t with
      tip m -> (tip n, m)
    | fork (l, r) ->
      let (t1, m1) = repmin l n
      and (t2, m2) = repmin r n
      in (fork (t1, t2),
        (if m1 < m2 then m1
          else m2));;
```

Bird as Two Functions

(Strict ML)

```
let rec rep t m = match t with
    tip n -> tip m
  | fork (l, r) ->
    fork (rep l m, rep r m);;

let rec tmin t = match t with
    tip n -> n
  | fork (l, r) ->
    let m1 = tmin l
    and m2 = tmin r
    in (if m1 < m2 then m1
      else m2);;

let bird t = rep t (tmin t);;
```

Theory of AGs =
transformation of functional programs

Bird as an AG

(declarations)

```
grammar TREE is
  START = start;
  start -> TREE;
  TREE = fork tip;
  fork -> TREE TREE;
  tip -> int;
end grammar;

attribute grammar
          bird (TREE): TREE is
  attribute
    synth $min (TREE): int;
    synth $tree (TREE): TREE;
    inh $n (TREE): int;
    synth $sttree (START): START;
```


Bird as an AG

(definitions)

```
where tip -> m use
  $min := m;
  $tree := tip ($n(tip));
end where;

where fork -> l r use
  $min := if $min(l) < $min(r)
    then $min(l)
    else $min(r) end if;
  $tree := fork($tree(l),
    $tree(r));
  $n(l) := $n; $n(r) := $n;
end where;

where start -> t use
  $sttree := start($tree(t));
  $n(t) := $min(t);
end where;
end grammar;
```


Comparison and Comments

- **In AGs, arguments and results are denoted by named attribute occurrences:**
  - * AGs are more verbose
    - **automatic generation of missing rules helps a lot**
  - * AGs are clearer:
    - **unrelated data flows are not mixed**
    - **support for incremental development**
- **An AG is a single specification**
  - * Analyses of AGs have a global scope
  - * New results on modularity and genericity makes specification easier while retaining this global scope
- **Evaluation order is automatically determined**
  - * When considered as a transformation of functional programs, this analysis is very powerful
- **BUT... AGs require a tree**
  - * How do you specify the factorial function?

Factorial as an AG
(very tentative syntax)

```
grammar couple FACT is
  dynamic
    FACT = fac_t fac_r;
    fac_t:FACT -> ;
    fac_r:FACT -> Fn_1:FACT;
end grammar;

attribute grammar factorial
   (FACT, $n:int): ($r:int) is
on FACT use
  if $n <= 1
  then with fac_t -> use
    $r := 1;
  end with;
  else with fac_r -> Fn_1 use
    $n(Fn_1) := $n - 1;
    $r := $n * $r(Fn_1);
  end with;
  end if;
end on; end grammar;
```


Grammar couples

- A **dynamic grammar**
  - * describes control flow (recursion scheme)
- A (possibly empty) **concrete grammar**
  - * describes a distinguished parameter: a physical tree
- An **association** between these grammars
  - * “natural” well-formedness conditions
  - * a same concrete node can be associated with several dynamic nodes

Dynamic AGs

- A **grammar couple**
- A collection of **attributes**
- A collection of **semantic rules blocks**
  - * Decision tree
    - – controlled by **conditions** over attribute values
    - – **dynamic productions** are at the leaves
    - – nodes contain **semantic rules** which are common to all the productions in the subtree below
- “Natural” well-formedness conditions

A DAG defines a function

- **Arguments**: values for inherited attributes of the root and, possibly, a concrete tree
- **Result**: values for the synthesized attributes of the root
- **Semantics**: “natural”

Describing Dynamic Semantics: The while example

Syntax

```
grammar couple Interpreter is
  ...
  concrete
    while:STAT ->
      cond:COND body:STAT;
  dynamic
    w_r=while:STAT ->
      cond=cond:COND
      body=body:STAT
      again=while:STAT;
    w_t=while:STAT ->
      cond=cond:COND;
  ...
end grammar;
```


The while example

Semantics

```
...
on while use
  $h.env(cond) := $h.env;
  if $val(cond) = bool-true
  then with w_r ->
          cond body again use
    $h.env(body) := $s.env(cond);
    $h.env(again) :=
    $s.env(body);
    $s.env := $s.env(again);
  end with;
  else with w_t -> cond use
    $s.env := $s.env(cond);
  end with;
  end if;
end on;
...
```


Other examples

(in papers)

- **Towers of Hanoi**
- **Double**
  - * **denotational semantics**
- **if-then-else**
- **etc.**

Implementation

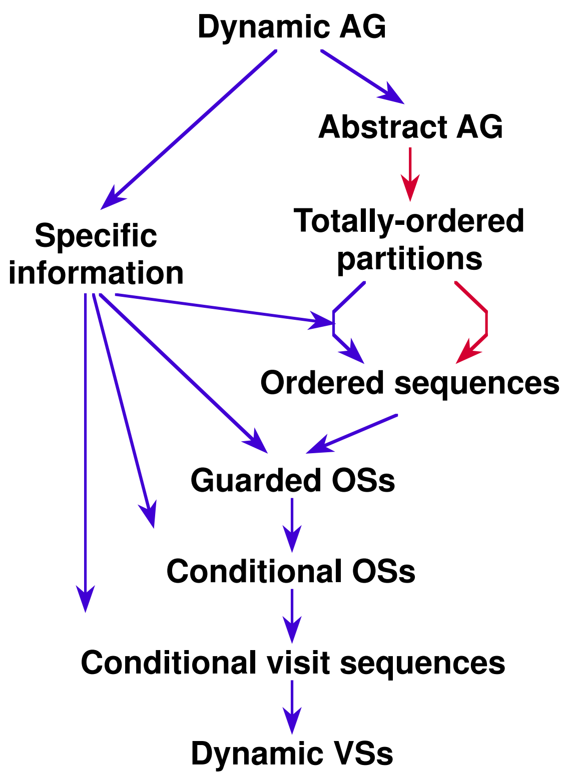
The “plane shift” principle

- **Method**
  - * **Build from a given DAG an “equivalent” classical AG**
  - * **Generate the evaluator for the latter using any traditional method**
  - * **Patch this evaluator to make it a correct implementation of the DAG**
- **Benefits**
  - * **Simplicity**
  - * **Reuse of (the core of) existing systems**
    - – a few man-weeks to implement DAGs in Fnc-2
  - * **Reuse of all existing implementation and analysis optimizations**
    - – control flow (evaluation order)
    - – memory optimisations
    - – incremental evaluation
    - – concurrent evaluation…

Why visit sequences?

- **Best trade-off between:**
  - * **expressive power**
    - – l-ordered class, SNC class
  - * **efficiency**
    - – time: completely deterministic, minimal overhead
    - – space: effective lifetime analysis techniques to store all the attributes out of the tree
  - * **versatility**
    - – incremental evaluation
    - – concurrent evaluation…
- **The basic paradigm used in Fnc-2**
  - * **reuse of system at hand**
- **Easiest to transform into functions**
  - * **gives a basis for our comparison and cross-fertilization of AGs and functional programs**

The generation process



The while example

Abstract AG

```
where [$val(cond) = bool-true /
       w_r]:STAT -> cond:COND
    body:STAT again:STAT use
  $h.env(cond) := $h.env;
  $h.env(body) :=
    dp($s.env(cond), $val(cond));
  $h.env(again) :=
    dp($s.env(body), $val(cond));
  $s.env := dp($s.env(again),
                $val(cond));
end where;

where [$val(cond) ≠ bool-true /
       w_t]:STAT -> cond:COND use
  $h.env(cond) := $h.env;
  $s.env :=
    dp($s.env(cond), $val(cond));
end where;
```


The while example

Ordered sequences

```
[val(cond) = bool-true / w_r]:
  h.env, h.env(cond),
  s.env(cond), val(cond),
  h.env(body), s.env(body),
  h.env(again), s.env(again),
  s.env
[val(cond) ≠ bool-true / w_t]:
  h.env, h.env(cond),
  s.env(cond), val(cond),
  s.env
```

The while example

Guarded ordered sequences

```
[val(cond) = bool-true / w_r]:
  h.env, h.env(cond),
  s.env(cond), val(cond),
   [val(cond) = bool-true],
  h.env(body), s.env(body),
  h.env(again), s.env(again),
  s.env
[val(cond) ≠ bool-true / w_t]:
  h.env, h.env(cond),
  s.env(cond), val(cond),
   [val(cond) ≠ bool-true],
  s.env
```


The while example

Conditional ordered sequence

```
h.env, h.env(cond),
s.env(cond), val(cond),
 [val(cond) = bool-true],
   { h.env(body), s.env(body),
   h.env(again), s.env(again),
   s.env },
   { s.env }
```


The while example

Conditional visit sequence

```
begin 1; eval h.env(cond);
visit 1, cond;
if { val(cond) = bool-true } then
  { eval h.env(body);
    visit 1, body;
    eval h.env(again);
    visit 1, again;
    eval s.env; leave 1; }
else
  { eval s.env; leave 1; }
```


The while example

Dynamic visit sequence

```
begin 1; eval h.env(cond);
visit 1, cond;
if { val(cond) = bool-true } then
  { eval h.env(body);
    visit 1, body;
    eval h.env(again);
    visit 1, again;
    eval s.env; }
else
  { eval s.env; };
leave 1;
```

From Dynamic VSs to Visit Functions

- Replace references to dynamic objects with references to their concrete counterparts, if any
- Check concrete parameter, if any, for the production that applies
- Pass values of conditions from the visit in which they are computed to the subsequent ones
  * non-temporary local attributes
- Apply modern techniques to store all the attributes out of the tree (what tree, anyway?)
  * Lifetime analysis
    - temporary attributes in global variables
    - temporary attributes in global stacks
    - non-temporary attributes in global variables
  * Binding trees

Cross-Fertilization

- There is a syntactic transformation between AGs and a subclass of (lazy) functional programs (Johnsson)
- This transformation is made much more broadly applicable by Dynamic AGs
- 30 years of research on AGs have yielded many good results on implementation and analysis
- How can these results be applied to functional programming via Dynamic AGs?
- Fertilization can also go the other way
- Other fertile directions:
  * AGs vs. logic programming
  * AGs vs. object-oriented programming

Example 1

Eager Evaluation of Lenient Programs

- When written as a single function, the Bird program requires lazy evaluation
  * “circularity” at top level
- When given the Bird AG, any evaluator generator will “resolve” the circularity and produce a two-pass, eager program equivalent to the “Bird as two (three) functions” one
- This idea is not new, but we have undertaken its systematic study, considered as a transformation or optimization of functional programs
- Preliminary experiments (Bird in Haskell) are encouraging

Example 2

Deforestation vs. Descriptional Composition of AGs

- Same goal: compose functions and optimize the result by not building the intermediate data structure
- Different methods
- Different domains of application
- How can one benefit from the other?

Other examples

- Attribute lifetime analysis for placement of function parameters in memory
- Attribute lifetime analysis for update-in-place optimizations
- Incremental evaluation
- Concurrent evaluation
- AG paradigms for modularity and reusability
  * Descriptional composition
  * Open AGs
  * Generic AGs
- etc.

Present and Future Work

- Dynamic AGs
  * Formaly define their semantics
  * Investigate the applicability of other classical AG evaluation methods
- Fnc-2
  * Strengthen the implementation of DAGs
  * Strengthen the system in general
  * Write lots of convincing applications
    – Revisit Ganzinger's translation of denotational semantics into AGs
- Cross-fertilization
  * Eager evaluation of lenient programs
  * Deforestation vs. descriptional composition
  * Update-in-place analysis and transformations
  * etc.

Conclusion

- Dynamic AGs
  * Simple extension that brings much expressive power
  * Retains all the benefits of AGs
    – Declarativity and other language features
    – Efficient implementation and accurate analysis
- Cross-fertilization
  * The new expressive power of DAGs makes well-known translations between AGs and other paradigms much more interesting
  * Use these translations to carry over good ideas
  * AGs have a lot of such good ideas!

Eager Bird in Haskell

```
main = program

data Tree a = Leaf a | Branch (Tree a) (Tree a)

calc :: Tree Int -> Int
calc l = case l of
   Leaf n -> n
   Branch left right ->
      let n1 = calc left
          n2 = calc right
      in if n1 < n2 then n1 else n2

replace t n = case t of
   Leaf m -> Leaf n
   Branch left right ->
      let t1 = replace left n
          t2 = replace right n
      in Branch t1 t2

mult_leave t = case t of
   Leaf n -> n
   Branch left right ->
      let p1 = mult_leave left
          p2 = mult_leave right
      in p1 * p2

creetree n = if (n == 1) then (Leaf n) else
    (Branch (creetree (n-1)) (creetree (n-1)))

program = let t = creetree 16
   in print (mult_leave (replace t (calc t)))
```

Lazy Bird in Haskell

```
main = program

data Tree a = Leaf a | Branch (Tree a) (Tree a)

repmin t n = case t of
   Leaf m -> (Leaf n, m)
   Branch left right ->
      let (t1, n1) = repmin left n
          (t2 , n2) = repmin right n
      in ((Branch t1 t2),
          (if  n1 < n2 then n1 else n2))

transb t = let (newtree, n) = repmin t n
   in newtree

mult_leave t = case t of
   Leaf n -> n
   Branch left right ->
      let p1 = mult_leave left
          p2 = mult_leave right
      in p1 * p2

creetree n = if (n == 1) then (Leaf n) else
   (Branch (creetree (n-1)) (creetree (n-1)))

program = let t = creetree 16
   in print (mult_leave (transb t))
```


Results

Non-optimized

	Lazy		Eager		Improvement	
height	time (sec)	space (Mb)	time (sec)	space (Mb)	time (%)	space (%)
15	0.64	3.21	0.58	2.10	9.38	34.69
16	1.36	6.42	1.26	4.20	7.35	34.69
17	3.04	12.85	2.94	8.39	3.29	34.69
19	13.12	51.38	12.22	33.56	6.86	34.69
21	63.02	205.53	55.56	134.22	11.84	34.69

Optimized

	Lazy		Eager		Improvement	
height	time (sec)	space (Mb)	time (sec)	space (Mb)	time (%)	space (%)
15	0.40	1.97	0.28	1.05	30.00	46.67
16	0.66	3.93	0.48	2.10	27.27	46.67
17	1.40	7.87	1.08	4.20	22.86	46.67
19	7.60	31.46	6.98	16.78	8.16	46.67
21	42.24	125.83	35.46	67.11	16.05	46.67

Comments

- **The only difference of “optimized” vs. “non-optimized” is the –O compiler flag**
- **Time and space figures are given by the profiler**
  * Some overhead may appear
  * Time figures are probably not very accurate (only one run per data point)
- **We’re absolutely not certain that the “eager” version is indeed evaluated eagerly!!!**
  * The data structure (output tree) built by the eager version may be exactly the same as that built by the lazy version, i.e., not fully evaluated
  * The improvements in time and space may be due only by the absence of tupling and untupling in the eager version
- **Still, 25% less time and 45% less space are impressive results…**