Dynamic Attribute Grammars : a declarative functional language

Didier Parigot

Didier.Parigot@inria.fr

http://www-rocq.inria.fr/∼parigot

http://www-rocq.inria.fr/oscar/www/fnc2/

Dynamic Attribute Grammars : a declarative functional language

Grammar $\neq$ Tree

Attribute $\neq$ Tree decoration

the classical view of attribute grammar is a particular case

Attribute grammars theory

Syntactic abstraction

Static analysis

Introduction

Attribute Grammars

Describe computations on structures

- A natural structural decomposition:
  the underlying grammar

- Declarative specification
  no explicit evaluation order

- Global static analysis
  Syntactic abstraction

- The poor expressive power:
  requires existence of a physical tree

The basic Ideas

- Grammar = Control flow
- Semantic rules = Data flow
- Recursive non-terminals
- Termination

The extensions of the input language

- Conditional productions
- Scheme productions

With the same global static analysis techniques

Outline of the talk

- **External view (input language)**
  Presentation by very simple examples

  (Attribute Grammar language)

  - Classical form (view)

  - Extensions

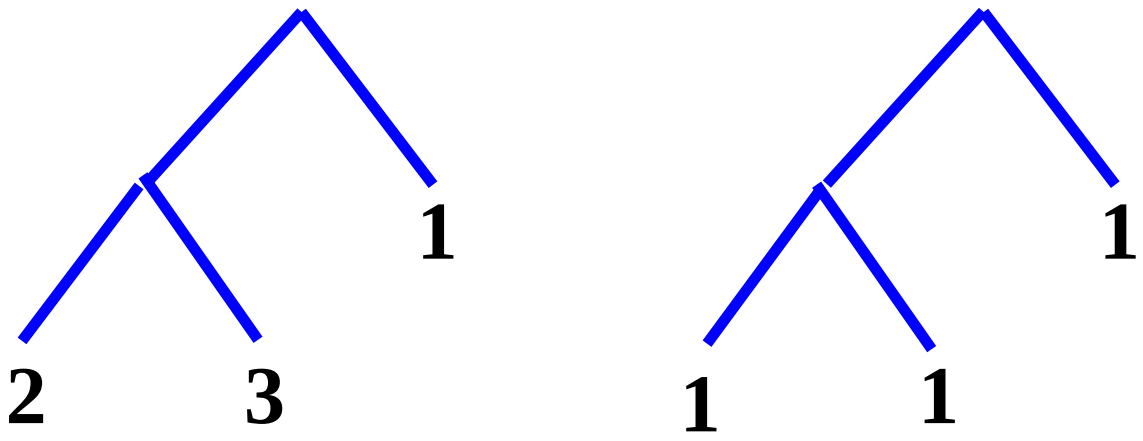
- **Internal view (evaluator generator view)**
  Why these extensions do not require modifications

  - Presentation of the FNC-2 system

- **Conclusion**

An example of tree transformation

([Bird] or [Johnsson87])



Abstract Syntax of Bird Transformation

```
AXIOM = axiom;
axiom -> TREE;

TREE = tip fork;
tip -> int ;
fork -> TREE TREE;

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

Bird with an Attribute Grammar

(attribute declarations)

```
attribute grammar bird (TREE in tree) :
 ($treeaxion:  TREE) is

attribute
 synthesized $min (TREE) : int;
 synthesized $tree (TREE) : TREE;
 inherited $n (TREE) : int;
 synthesized $treeaxiom (AXIOM) : AXIOM;
```

Bird with an Attribute Grammar

(attribute definition)

```
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(fork);
 $n(R) := $n(fork);
end where;

where axiom -> T use
 $treeaxiom := axiom($tree(T));
 $n(T) := $min(T);
end where;
end grammar;
```

Conditional productions

```
where fork -> L R  use
 $min := if ($min(L) <= $min(R))
      then $min(L) else $min(R) end if;
```

rewritten in

```
where fork[$min(L) <= $min(R) ] -> L R  use
 $min := $min(L);
end where;
where fork[] -> L R use
 $min := $min(R);
end where;
```

Grammar = Control flow

Conditions = Termination control

Scheme productions

allow to describe recursion schemes independently from any physical structure

```
FACT = with scheme

  fac-t -> ;
  fac-r -> FACT;
end scheme;
  attribute grammar FACT($n:int):($r:int) is


 where fac-t [$n <= 1] -> use
  $r := 1;
 end where;

 where fac-r [] -> F use
  $n(F) := $n - 1;
  $r := $n * $r(F);
 end where;
```

Other examples

```
HANOI = with scheme
 hr -> HANOI HANOI;
 ht ->;
end scheme;
```

Describing dynamic semantics

```
Expr = ...  while ...;
while -> c:Cond e:Expr
 with scheme
  wt -> c;
  wr -> c e wr;
 end scheme;

where wt[$c (c) = false] -> c use ...

where wr[] -> c e w use ...
```

Other examples

```
STMT = double ...  ;
double -> p:STMT
 with scheme
  dd -> p p ;
 end scheme;

EXPR = ifthenelse ...;
ifthenelse -> c:EXPR t:EXPR e:EXPR
 with scheme
  ifthen -> c t;
  ifelse -> c e;
 end scheme;
```

Intrinsic expressive power

- Primitive recursion
  First-order functional language

- Structural and declarative method.

- Supports incremental development
  by attributes

- Static analysis

Internal view

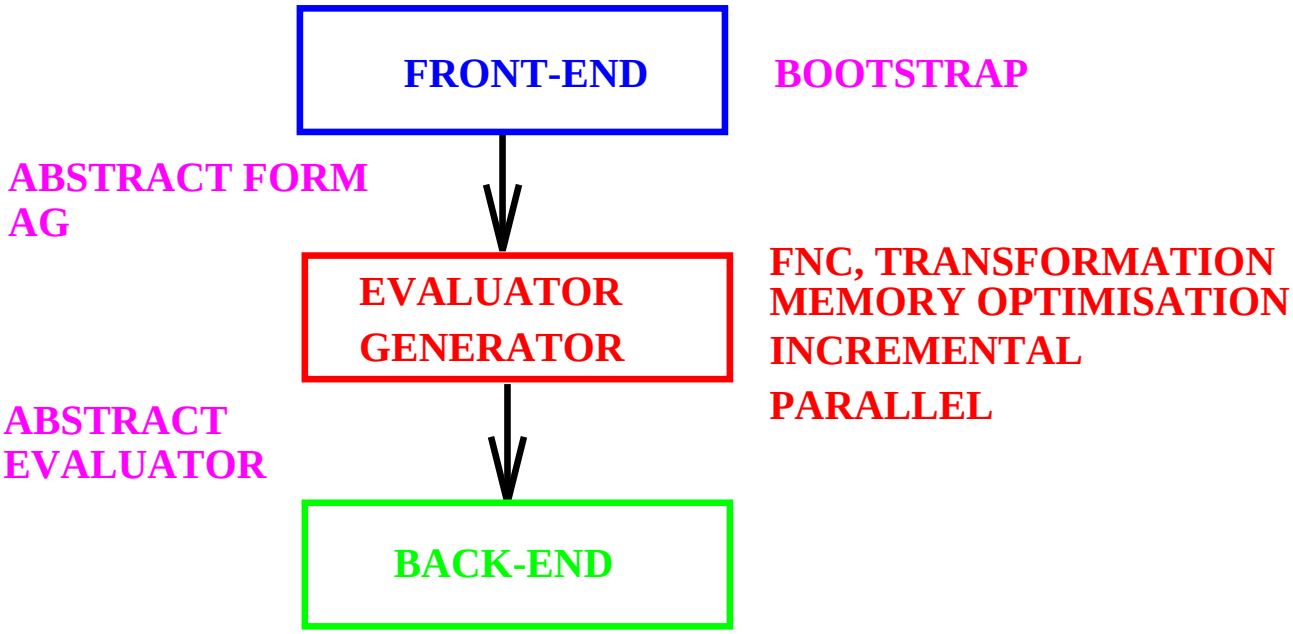
An implementation into the FNC-2 system required *only a few man-weeks: Bootstrap*

- Decomposition of the FNC-2 system
- Transform new AGs into classical AGs
  Same evaluator generator
- Memory optimizations:
  Store attribute outside tree
  SNC Class (Strongly Non-Circular)

Dynamic attribute grammars

FNC-2

OLGA



(C[Bootstrap], LISP[Centaur], fSDL[COSY], ML[caml])

The formal definitions, the theorems and their proofs appear in [Parigot96, PLIPL'96]

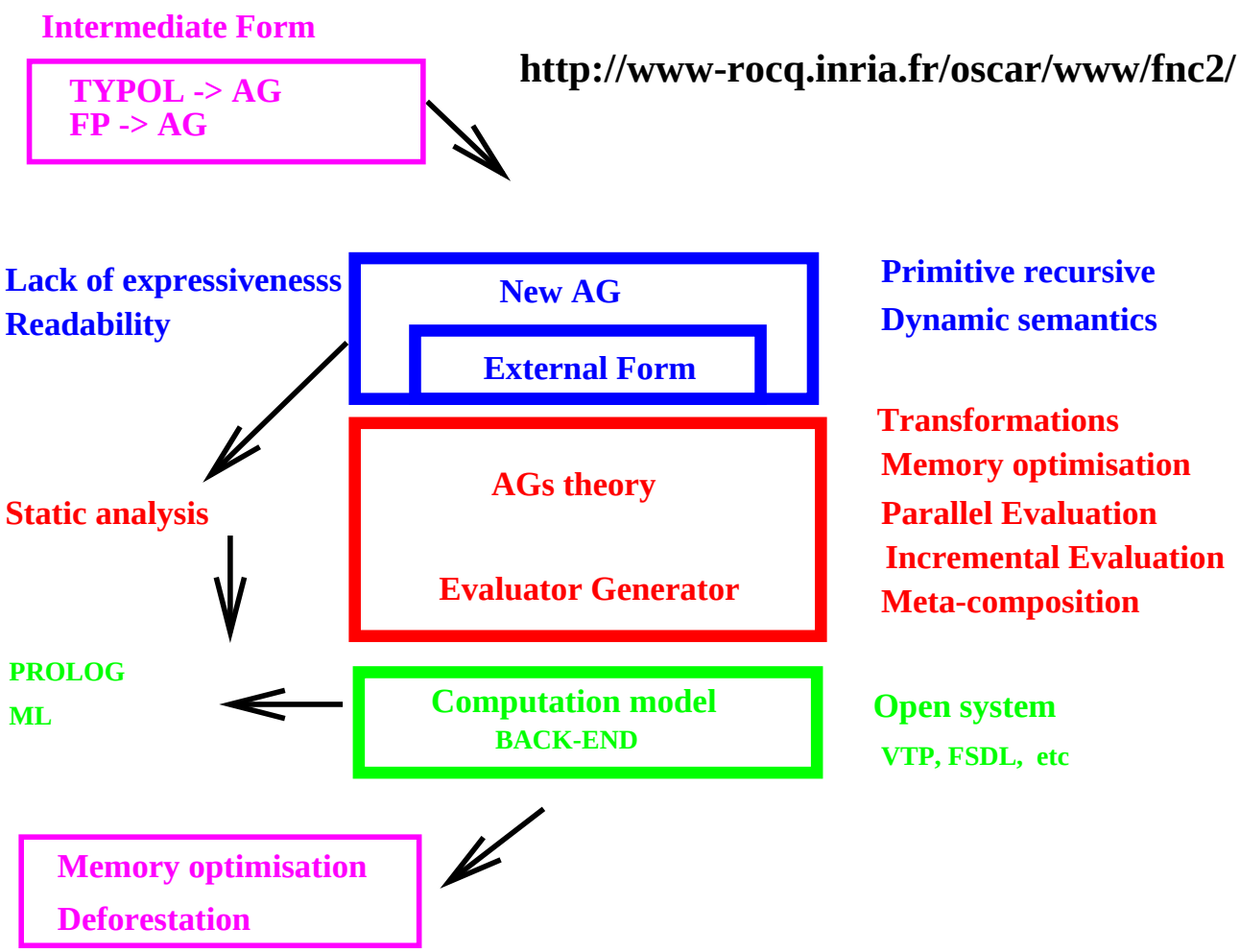
a factored form

```
where while -> C E use
 $h.env(C) := $h.env(while);
 if [$c(C) = false] then
   scheme wt -> C use
    $s.env(wt):= $h.env(wt)
   end scheme
 else
  scheme wr -> C E W use
   $h.env(E):= $h.env(wr);
   $h.env(W):= $s.env(E);
   $s.env(wr):= $s.env(W);
  end scheme;
end where;
```

a flattened form

```
where wt[$c (C) = false] -> C use
 $h.env(C) := $h.env(while);
 $s.env(wt):= $h.env(wt);
end where;
where wr[] -> C E W use
 $h.env(C) := $h.env(while);
 $h.env(E):= $h.env(wr);
 $h.env(W):= $s.env(E);
 $s.env(wr):= $s.env(W);
end where;
```

The main advantages of this new approach

- Intrinsic expressive power (dynamic semantic)
- Static analysis
- Cross-fertilization
- Transformation : Lenient/lazy evaluation
- Memory Optimization : global variables
- Descriptional composition : Deforestation
- Incremental evaluation
- Parallel evaluation

Dynamic Attribute Grammars