Attribute Grammars :
a Declarative functional Language

Didier Parigot

Didier.Parigot@inria.fr

http://www-rocq.inria.fr/∼parigot

http://www-rocq.inria.fr/charme/FNC-2/

Attribute Grammars: a Declarative functional Language

Grammar $\neq$ Tree

Attribute $\neq$ Tree Decoration

the Classical view of AG is a Particular Case

Attribute Grammars Theory

Syntactic Abstraction

Static Analysis

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

External View (Input Language)
Presentation by very simple examples

(written in ML and AG language)

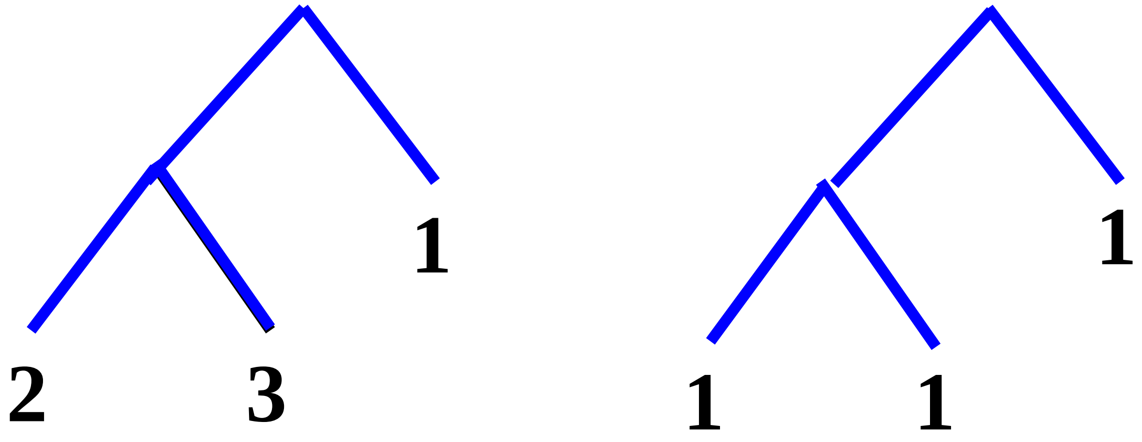
- Classical Form (view)
- Extensions

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 Johnsson)



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 one ML function

```
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 with two (three) ML functions

```
let rec replace t m = match t with
  tip (n) -> tip(m) |
  fork(L, R) ->
   fork ((replace L m), (replace R m));;

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

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

Theory of AGs :Transformation of Functional programs

SNC class (strongly non-circular

INRIA

Bird with an Attribute Grammar (declarations)

```
attribute grammar bird (TREE) : (TREE) is
attribute
 synthesized $min (TREE) : int;
 synthesized $tree (TREE) : TREE;
 inherited $n (TREE) : int;

synthesized $treeaxiom (AXIOM) : AXIOM;
```

Bird with an Attribute Grammar (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;
```

- The input language of AGs is more verbose!!!

  Arguments and Results are denoted by Attribute Occurrences

```
f (..., expression, ...) =>
$attribute (f):= expression;
```

- AG programming : a single specification.

- Transformations : Static Analysis.

Conditional Productions

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

```
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 graph

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 ...
```


INRIA

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

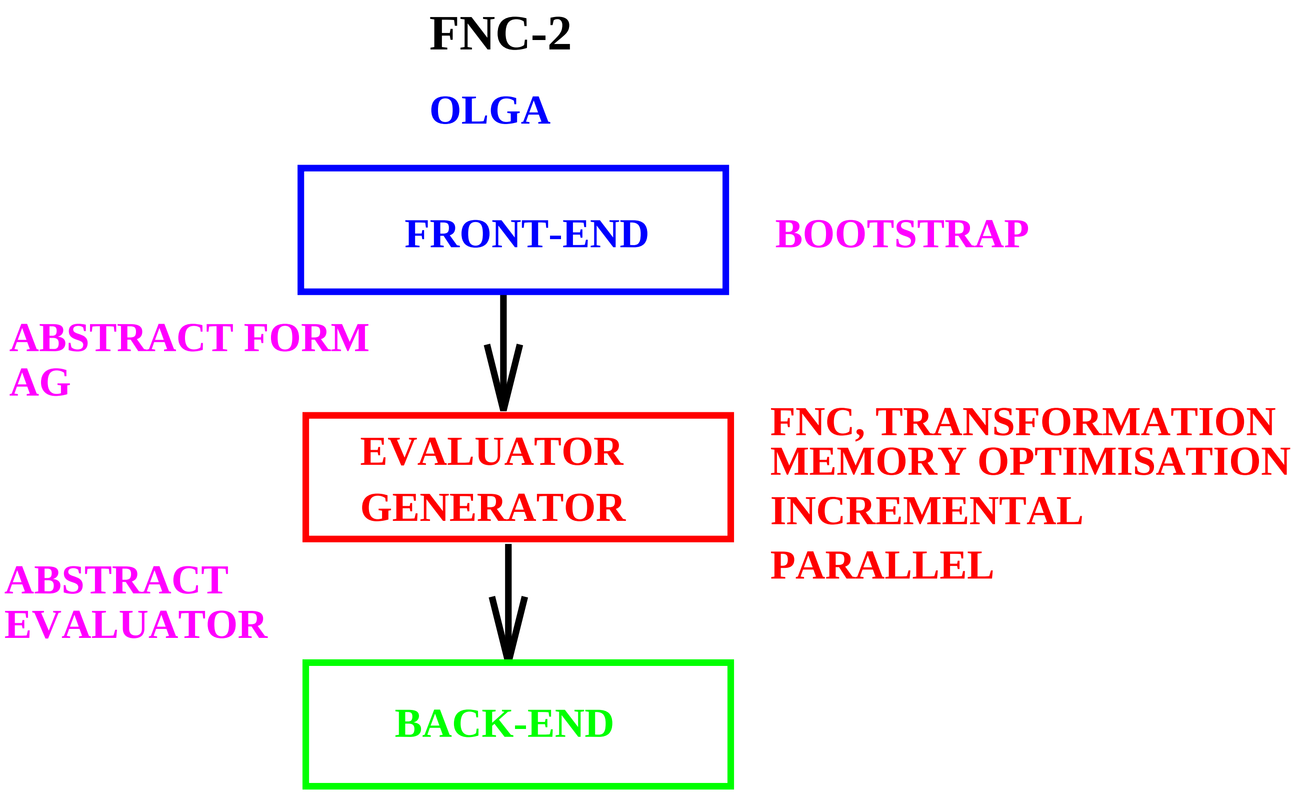
- Primitive Recursion.
- Structural and Declarative method.
- Supports incremental development by attributes
- Static Analysis.

Internal View

A 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 Optimisations:
  Store attribute outside the tree
  SNC Class (Strongly Non-Circular)

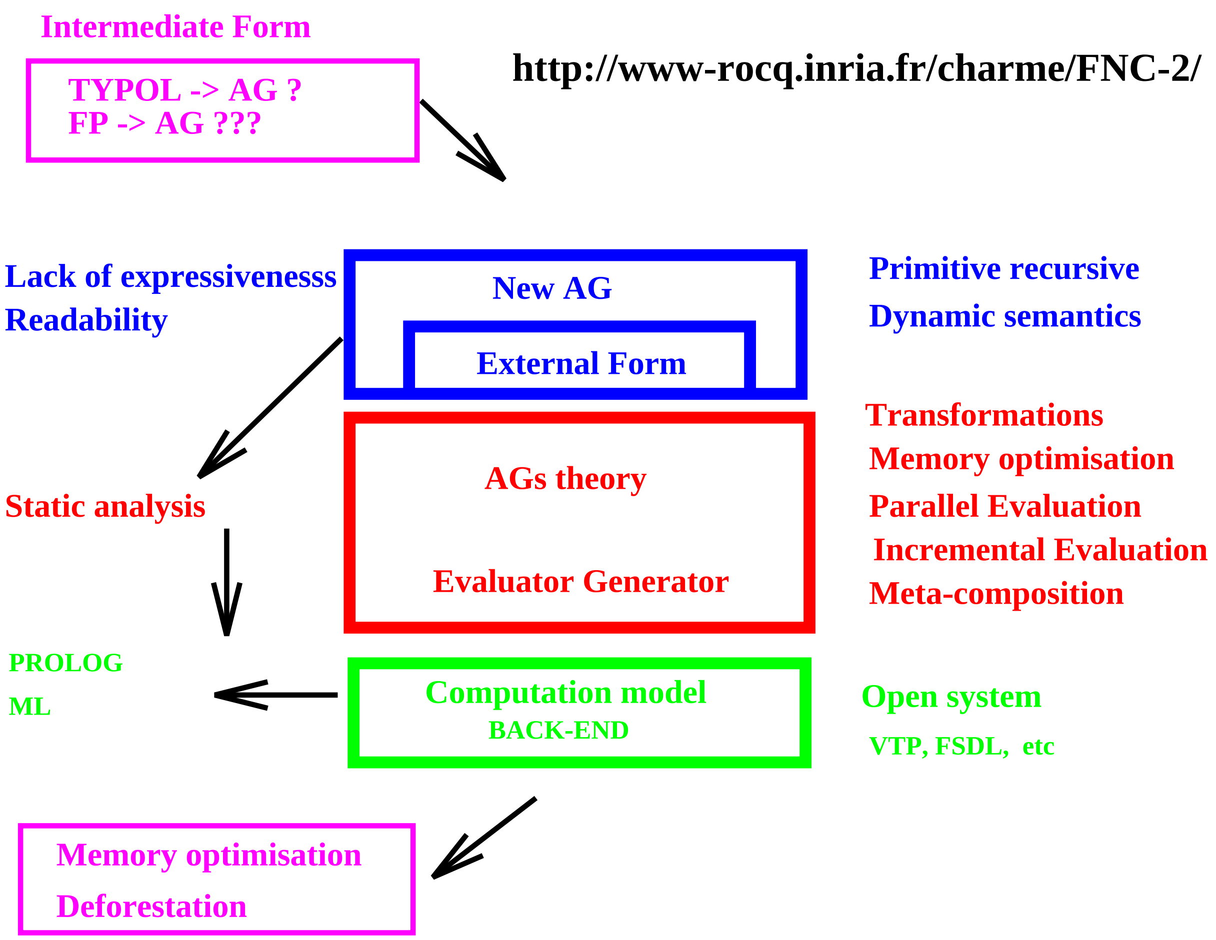
Presentation of the FNC-2 system



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

The main Advantages of this new approach

- Intrinsic expressive power.
- Cross-fertilization



INRIA