Decorated Attribute Grammars

Attribute Evaluation Meets Strategic Programming

CC 2009, York, UK

Lennart Kats, TU Delft (me)
Tony Sloane, Macquarie
Eelco Visser, TU Delft

March 19, 2009

Software Engineering Research Group

TU Delft
Delft University of Technology
Context

- Domain-specific languages
  - example: WebDSL
  - language composition and extension
- SDF/SGLR + Stratego/XT
  - abstract syntax trees
  - traversal, rewrite rules
Trees and Attribute Grammars

- Attributes
  - Declarative, compositional equations
  - Express dependencies between nodes
- Attribute evaluator
  - Determines evaluation order
Basic Example: Global Minimum

- Synthesized: flows up

```python
def Root(t):
    id.min := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)

def Leaf(v):
    id.min := v
```
Basic Example: Global Minimum

- Synthesized: flows up
- Inherited: flows down

```python
def Root(t):
    id.min := t.min
    t.gmin := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)
    t1.gmin := id.gmin
    t2.gmin := id.gmin

def Leaf(v):
    id.min := v
```
Basic Example: Global Minimum

```python
def Root(t):
    id.min := t.min
    t.gmin := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)
    t1.gmin := id.gmin
    t2.gmin := id.gmin

def Leaf(v):
    id.min := v
```
def Root(t):
    id.min := t.min
    t.gmin := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)
    t1.gmin := id.gmin
    t2.gmin := id.gmin

def Leaf(v):
    id.min := v
Global Minimum: Identifying Copy Rules

```python
def Root(t):
    id.min := t.min
    t.gmin := t.min

def Fork(t1, t2):
    id.min := min(t1.min, t2.min)
    t1.gmin := id.gmin
    t2.gmin := id.gmin

def Leaf(v):
    id.min := v
```
Introducing Decorators

• Abstract over traversal or evaluation pattern
  • Express intent
    • min: “upward flow”
    • gmin: “downward flow”
  • May introduce default behavior
  • May modify existing behavior
Example: up/down copying decorators

```python
def Root(t):
    t.gmin := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)

def Leaf(v):
    id.min := v
```
Example: up/down copying decorators

```python
def Root(t):
    t.gmin := t.min

def Fork(t1, t2):
    id.min := <min> (t1.min, t2.min)

def Leaf(v):
    id.min := v

def down gmin

def up min
```
Introducing Decorators (ct'd)

Based on *strategic programming*

- **Programmable**: decorators available as a library
- **Generic**: independent of a particular tree
- **Reflective**: may use properties of attributes

```plaintext
decorator down(a) =
  if a.defined then
    a
  else
    id.parent.down(a)
  end

decorator up(a) =
  if a.defined then
    a
  else
    id.child(id.up(a))
  end
```
Basic Building Blocks of Decorators

**Arguments**
- attribute $a$
- functions, terms

```plaintext
decorator \texttt{down}(a)
\text{if } a.\texttt{defined} \text{ then}
    \begin{itemize}
    \item $a$
    \end{itemize}
\text{else}
    \begin{itemize}
    \item \texttt{id.parent.down}(a)
    \end{itemize}
\text{end}
```

**Reflective attributes**
- $a.\texttt{defined}$
- $a.\texttt{attribute-name}$
- $a.\texttt{signature}$

**Tree access attributes**
- \texttt{id.parent}
- \texttt{id.child}(c), \texttt{id.prev-sibling}, ...

**Recursion**
Abstraction Using Decorators (1)

Boilerplate code elimination:
• avoid repetitive code (e.g., “copy rules”)
• reduce accidental complexity
• implement some of the boilerplate-coping mechanisms of other AG systems
Abstraction Using Decorators (2)

Control over evaluation:
- **tracing**
- memoization
- assertions

```python
def trace gmin

@decorator
def trace(a) =
    t := id;
    a;
    log(||[a.attribute-name, " at ", t.path, ": ", id])
```
Abstraction Using Decorators (2)

Control over evaluation:
- tracing
- memoization
- assertions

```plaintext
default-caching(a)
    if id.get-cached(a) then
        id.get-cached(a)
    elsif a then
        ...
        a;
        ...
        set-cached...
    end
```
Abstraction Using Decorators (2)

Control over evaluation:
- tracing
- memoization
- assertions

```python
def assert-after(<leq> (id.gmin, id.min)) gmin

decorator assert-after(a, c) =
    t := id;
a;
id.set-cached-for(a|t);
if not(<c> t) then
    fatal-err(||"Assertion failed for ",
             a.attribute-name, " at ", t.path)"
end
```
Abstraction Using Decorators (3)

Help in typical compiler front-end tasks:
• name analysis
• type analysis
• control- and data-flow analysis

⇒ encapsulation of recurring attribution patterns
Type Analysis with Aster

```python
def type:
    Int(i) → IntType
    [[ var x : t; ]] → t
    Var(x) → id.lookup-local(|x).type
    [[ f(arg*) ]] → id.lookup-function(|f, arg*).type
    ...
```

Concrete syntax [Visser 2002]
- `VarDecl(x, t)`

Reference attributes [Hedin 2000]
- look up declaration nodes

```javascript
var x : Int;
```
Type Analysis with Aster: Using Decorators (1)

Lookup decorators require:
- Lookup type (ordered, unordered, global, ...)
- Tree nodes to fetch
- Scoping definition

```python
def lookup-ordered(id.is-scope) lookup-local(|x|):
    [[ var x : t; ]| → id
    [[ x : t ]| → id
```
Type Analysis with Aster: Using Decorators (2)

Lookup decorators require:
- Lookup type (ordered, unordered, global, ...)
- Tree nodes to fetch
- Scoping definition

```python
def is-scope:
    [[ { s* } ]] → id
    [[ function x(param*) : t { stm* } ]] → id
    ...
```
Type Analysis with Aster: Using Decorators (3)

```python
def lookup-unordered(id.is-scope) lookup-function(|x, arg*):  
[[ function x (param*) : t { stm* } ]]| → id  
where  
argtype* := arg*.map(id.type);  
paramtype* := param*.map(id.type);  
paramtype*.eq(|argtype*)
```
Type Analysis with Aster: Decorator Definitions

**decorator** down lookup-ordered(a, is-s) =
if a then
  a
else
  id.prev-sibling(id.lookup-inside(a, is-s))
end

**decorator** lookup-inside(a, is-scope) =
if a then
  a
else if not(is-scope) then
  // enter preceding subtrees
  id.child(id.lookup-inside(a, is-scope))
end

**function** x() : Int {
  var j : Int;
  ...}

**function** y() : Int {
  if (true) {
    var i : Int;
  }
  return j;
}
Error Reporting Using Decorators

```python
def errors:  
[ while (e) s ]] → "Condition must be of type Boolean"
  where not(e.type ⇒ BoolType)
...

def collect-all add-error-context errors

@add-error-context

def add-error-context(a) =
  <conc-strings > (a," at ", id.pp, " in ", id.file, ":",
                  id.linenumber)
```

*module Constraints*

*module Reporting*

Decorator stacking
Control-flow Analysis (1)

```python
def default(id.default-succ) succ:
    [[ if (e) s₁ else s₂ ]] → [s₁, s₂]
    [[ return e; ]] → []
    [[ { s₁; s* } ]] → [s₁]

...  

decorator default(a, default) =
    if a.defined then
        a
    else
        default
    end
```
Control-flow Analysis (2)

```python
def down default-succ:
    Program(_) → []
    [s₁, s₂ | _].s₁ → [s₂]
    ...
```
“But Wait, There's More!”

• Data-flow analysis
  • reaching definitions, liveness, ...
  • data-flow equations as attribute equations
  • circular (fixed point) attribute evaluation
• Deriving the reverse control flow

// Statement copies itself to successors

def contributes-to(id.succ) pred:
  stm → id
## Go Figures!

<table>
<thead>
<tr>
<th>Category</th>
<th>Modules</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>3</td>
<td>341</td>
</tr>
<tr>
<td>Compiler</td>
<td>25</td>
<td>2831</td>
</tr>
<tr>
<td>Library</td>
<td>25</td>
<td>1845</td>
</tr>
<tr>
<td>Tests</td>
<td>33</td>
<td>1531</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>86</td>
<td>6548</td>
</tr>
</tbody>
</table>
Concluding Remarks

• Language growth in the hands of the users
  • decorators implement automatic copy rules, self rules, collection attributes, circular attributes, ...
• Combine generic behavior with (simple) reflection
• Future:
  • Library, language extension
  • IDE integration

http://www.strategoxt.org/Stratego/Aster