Parameterized reference attributes: examples and properties

Görel Hedin & Emma Söderberg
Computer Science, Lund University

IFIP WG 2.11 meeting, Halmstad, June 25, 2012

Parameterized attributes are part of reference attribute grammars.

Agenda

- Background
  - Reference Attribute Grammars
  - JastAdd
- Examples of parameterized attributes
  - Name analysis, Type checking, …
- Properties
  - Incremental evaluation, …
- Ongoing work
- Conclusions

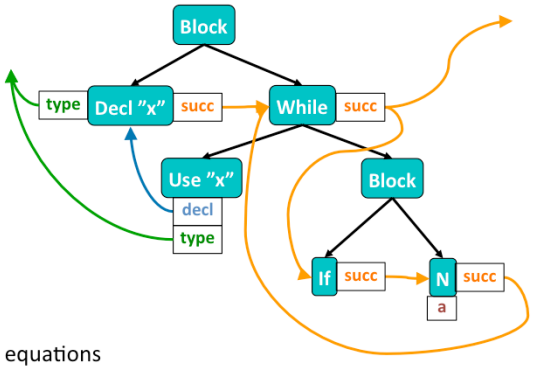
G. Hedin: Reference Attributed Grammars. Informatica (Slovenia) 24(3): (2000). In http://www.informatica.si/PDF/Informatica_2000_3.pdf

Reference Attribute Grammars

AST: abstrac syntax tree
[Knuth68]: Attributes are computed properties of nodes, defined by equations:
T N.a = *expr without visible side-effects*
  defined in node: *synthesized*
  defined in parent: *inherited*

[Hedin00]: Attributes can be references:
  Decl Use.**decl** = …
  Set<Stmt> Stmt.**succ** = …

Remote access via references:
  Type Use.**type** = **decl**.**type**

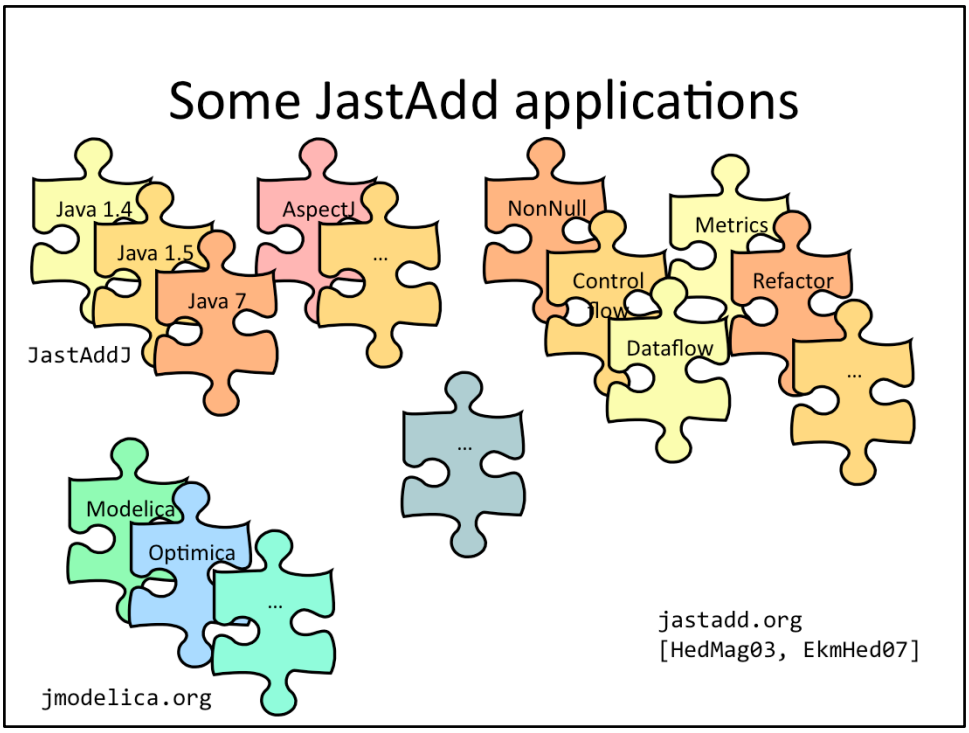


Declarative implementation
Extensible: just add more attributes and equations
Attributes evaluated on demand
Attributes cached for efficiency

Attributed AST is the full program model
JastAdd: metacompilation tool based on RAGs

Reference attribute grammars are an extension to Knuth's attribute grammars (Knuth 68)
Knuth's attribute grammars include synthesized and inherited attributes.
Reference attribute grammars extend Knuth's grammars with reference attributes and parameterized attributes.

Note that:
-Graphs can be defined on top of the AST
-The graphs can be cyclic
-Data structures needed during compilation can be modelled as attributes on the AST.
-Attributes are evaluated on demand, and cached for efficiency. For example, if the attribute Use.type is demanded, the right-hand side of its equation is evaluated (decl.type), and in order to do so, the decl attribute is first evaluated, evaluating the right-hand side of its equation, and so on.



JastAdd is a metacompilation system implementing RAGs.

A full Java compiler has been implemented in JastAdd (JastaddJ). (Ekman and Hedin, OOPSLA 2007)
A key advantage of RAGs and JastAdd is the strong support for modularization.
For example extending the Java 1.4 compiler with modules for Java 5 and Java 7.

JastAddJ has been extended both with language extensions and with different analyses.

Another large application of JastAdd is an open-source compiler for Modelica, called JModelica.org.
Modelica is a language for modeling and simulation of physical systems.
Optimica is an extension to Modelica to describe optimization problems for physical systems.
Optimica has been implemented as a modular extension to Modelica, using JastAdd. (Hedin, Åkesson, Ekman, IEEE Software 2011)

Attribution mechanisms

- Synthesized attributes [Knu68]
- Inherited attributes [Knu68]
- Reference attributes [Hed00]
- Parameterized attributes [Hed00, Ekm06]
- Broadcasting [Ekm06]
- Rewrites [EkmHed04]
- Nonterminal attributes (higher-order) [VogSwiKui89]
- Circular attributes [Far86, MagHed07]
- Collection attributes [Boy92, MagEkmHed09]

JastAdd supports several attribution mechanisms.
This talk focuses on parameterized attributes.

Parameterized attributes

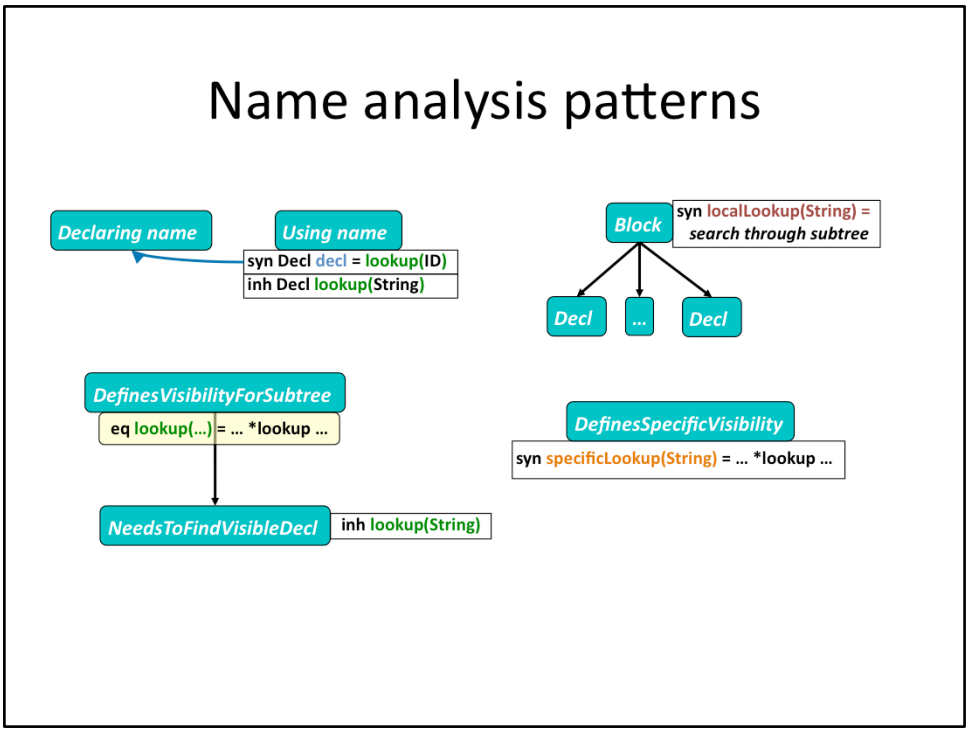
Attributes can have parameters:
T N.**a(T1 p1, T2 p2)** = …

A parameterized attribute has one element for each possible params-tuple.

The number of elements is (usually) unbounded.

Only demanded elements are computed.

A parameterized attribute is similar to a method (without side-effects).
But it differs in that the results are cached (memoized).
And it differs in that circularity is checked dynamically. (If an attribute depends on itself, this is reported as an error at evaluation time. Unless the attribute is explicitly defined as circular in which case it is evaluated iteratively until a fixed point is reached.)
And if the attribute is inherited (in the attribute-grammar sense), the equation/ implementation is not located in the same node, but instead in a parent of the node.



For illustration of parameterized attributes, we take a look at name analysis.

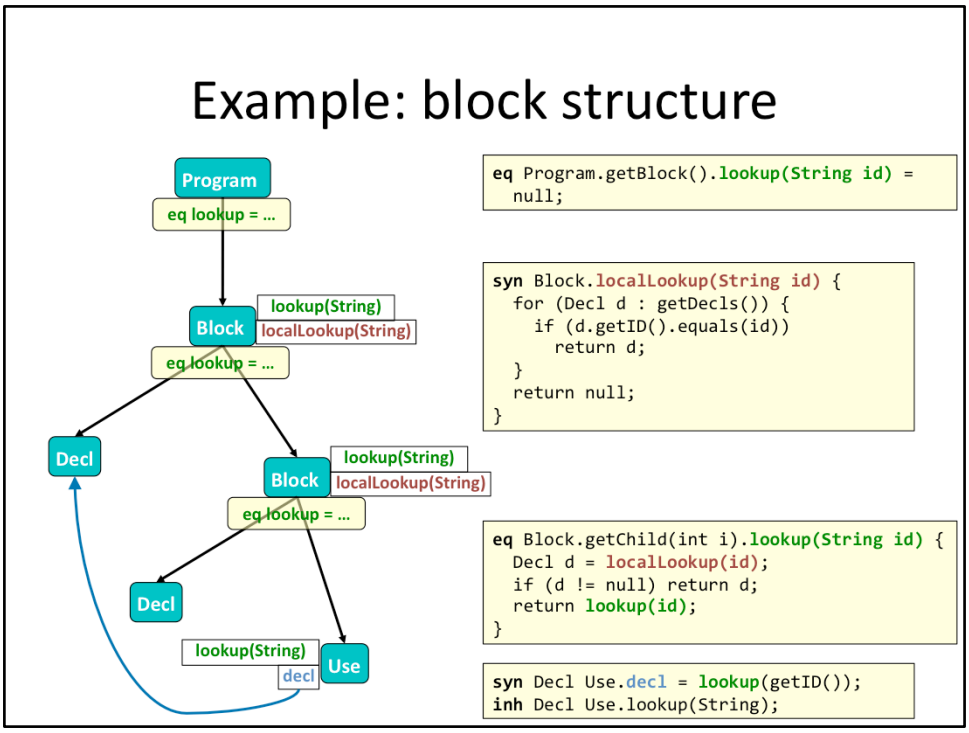
In RAGs, you typically use some small name analysis patterns that are combined to implement name analysis for a specific language.

For nodes that *Use* names, you add a *decl* attribute that will refer to the appropriate *declaration*.
The *decl* attribute is defined using an inherited attribute *lookup*, that is parameterized.
Lookup will return a reference to the appropriate Decl node. But exactly how that is done is delegated to the context, by declaring lookup as an inherited attribute.

The parent defining the lookup attribute typically will do so by delegating to other attributes that are called lookup or similar.

A language construct with local declarations, i.e., a *block*, defines an attribute typically called *localLookup* that searches a local part of the AST to find an appropriate declaration node.

There can also be additional *specific lookup* attributes, that delegate to other lookup attributes. This can be used, e.g., to implement inheritance in object-oriented languages. These specific lookup attributes are typically synthesized, in contrast to the inherited *lookup* attribute.

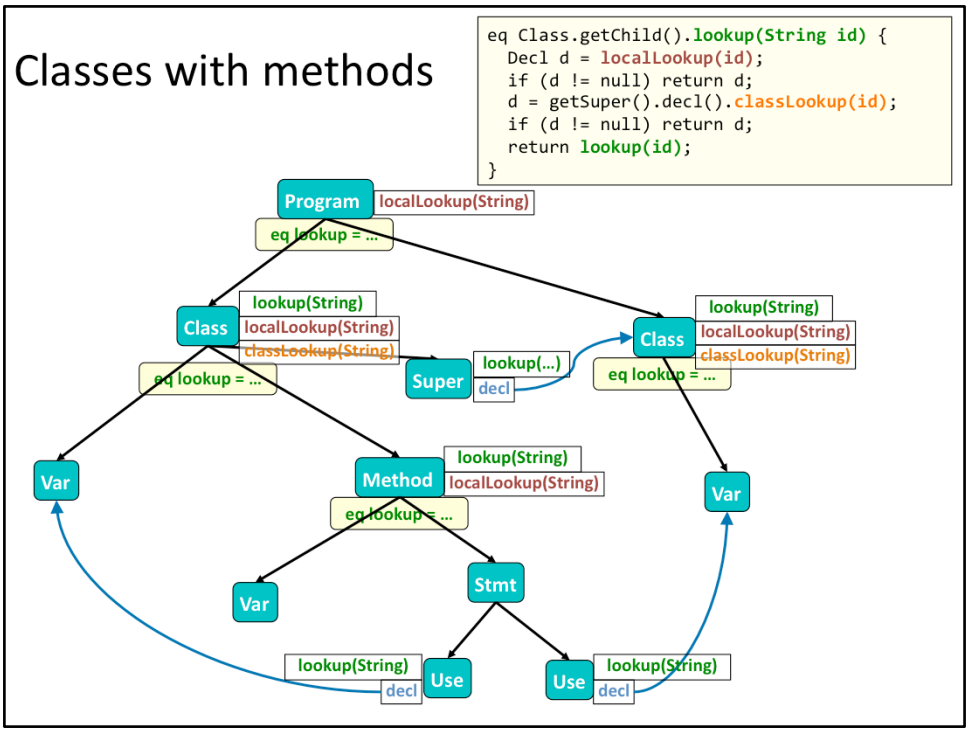


Here is a concrete example of applying the patterns: implementing name analysis for a simple block structured language.

The Use node has a decl attribute that is defined by calling the inherited lookup attribute with the Use's name (getID) as a parameter.

The lookup attribute of the Use node is defined by the parent Block, by first checking if the declaration is among the local declarations, and if not, returns the value of its own lookup attribute for the same parameter, which in turn is defined by its Block parent.

The localLookup attribute of a Block, simply traverses the local declarations and returns a reference to the appropriate declaration, if found. Otherwise, null is returned.



Here is an example of applying the name analysis patterns for a language with both block structure and inheritance.

Most of the attributes are similar to the ones in the previous example.

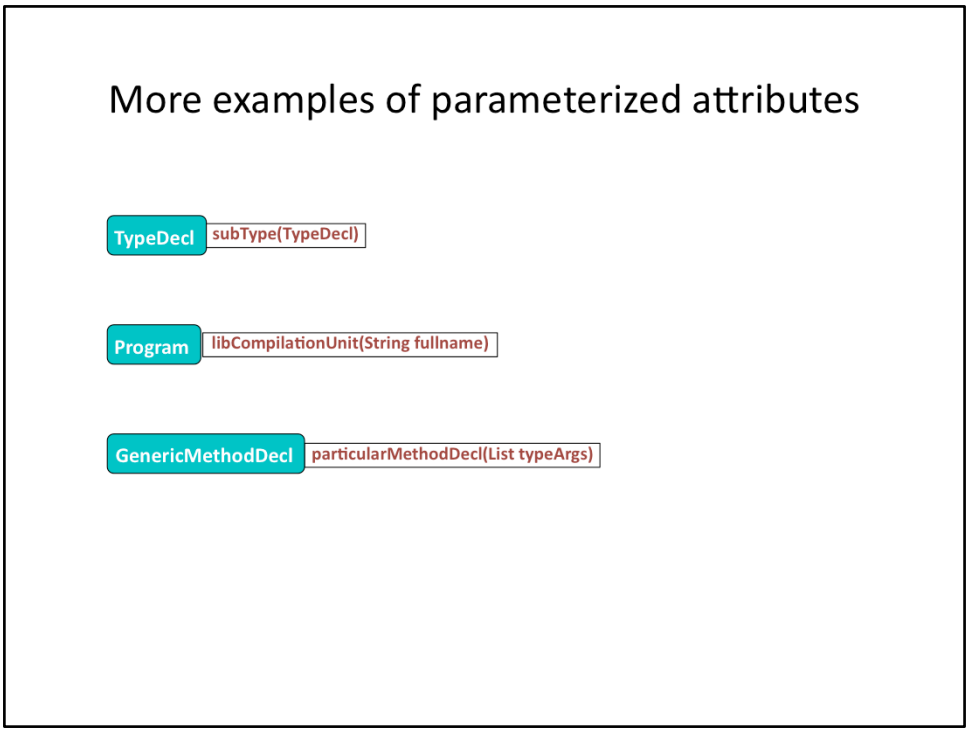
Both Classes and Methods have localLookups to find their local declarations.

Classes additionally have a classLookup attribute. This is a specific lookup attribute that combines the local declarations with the declarations found in the superclasses.

The superclass is located via the decl attribute of Super. Super is actually a kind of Use node: it uses the name of the superclass. So we add decl and lookup attributes to Super, and this gives us a reference to the superclass.

The Class.lookup equation simply combines the local lookup, the classLookup and its own inherited lookup attribute to define visibility for Uses inside the class.

Note, the code above is just a sketch. For a complete running example, see the PicoJava example at the JastAdd.org site.



Here are some more examples of parameterized attributes.

Types are represented by nodes in the syntax tree.
With the parameterized attribute subtype, you can ask one type node if it is a subtype of another type.
This attribute is typically implemented with double dispatch in order to elegantly compare related kinds of types.

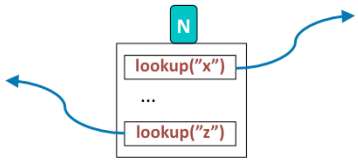
libCompilationUnit is a so called nonterminal attribute (higher-order attribute).
A nonterminal attribute is an attribute that has a syntax tree as its value (Vogt, Swierstra, Kuiper, 1989)
The attribute Program.libCompilationUnit(String fullname) returns an AST for the compilation unit for the file corresponding to the compilation unit named ”fullname”.
This tree is constructed by parsing that file.

The attribute GenericMethodDecl.particularMethodDecl(List typeArgs) is also a parameterized nonterminal attribute that returns a method declaration that is specialized with a list of specific type arguments.

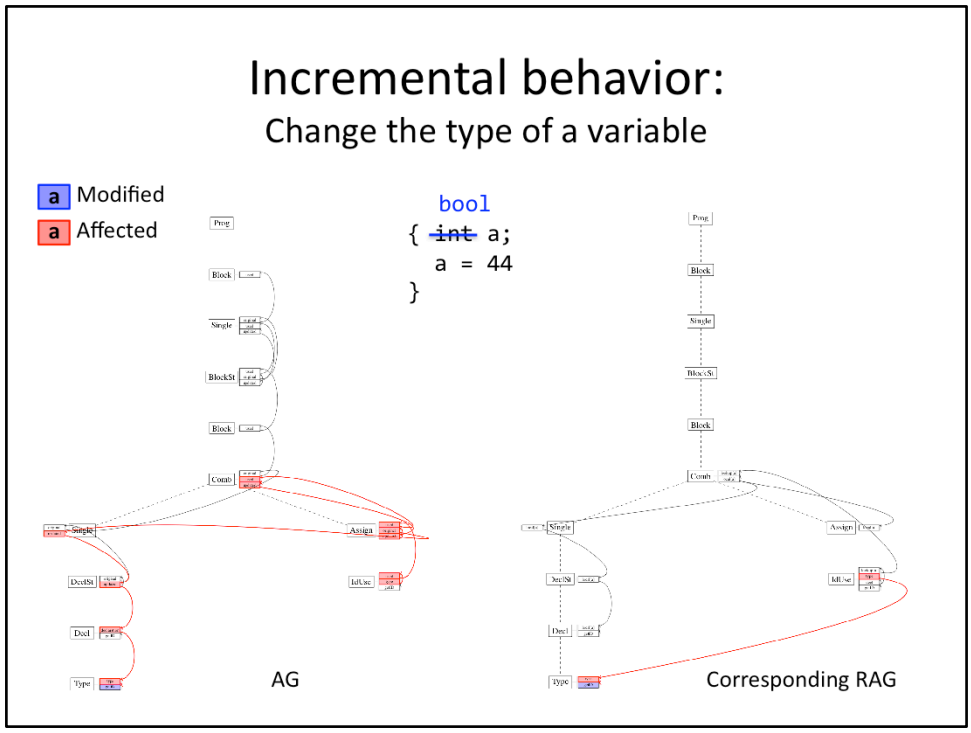
Parameterized reference attributes

The value of each attribute is small:
often just a single reference

Enables computed properties (attributes)
to be defined in a very finegrained way



Because of each element only defining a small piece of information, this makes parameterized attributes very suitable for incremental computations. As will be shown on the following slides.



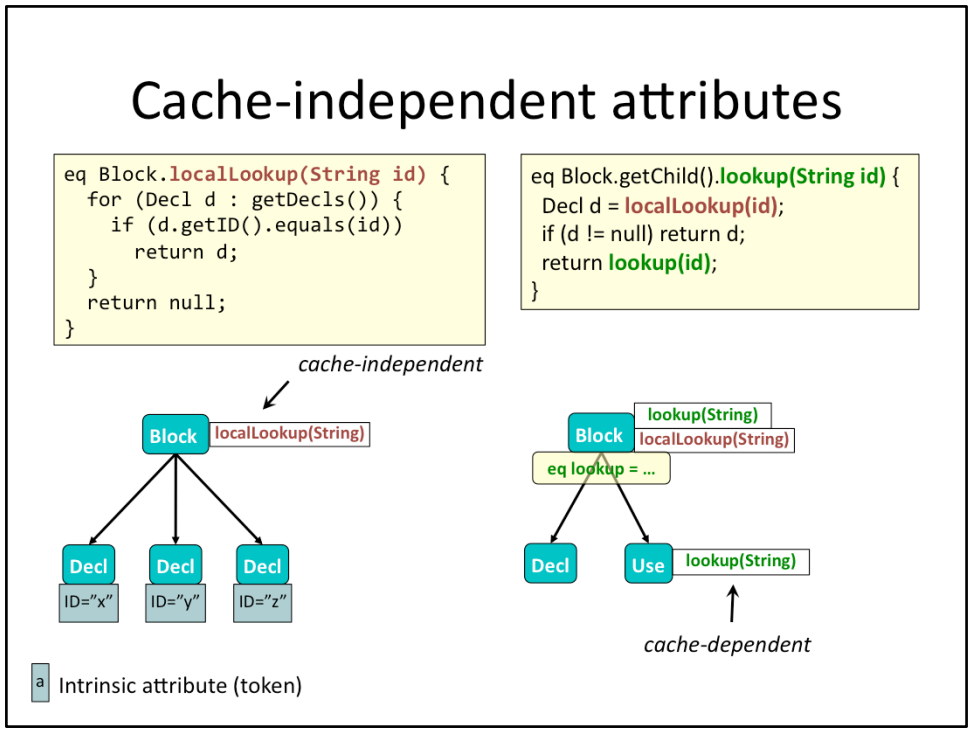
A modification of a declaration leads to many affected attributes in an ordinary AG.
A corresponding RAG has much fewer affected attribute.

Affected attribute means attribute with a new value.

AGs versus RAGs

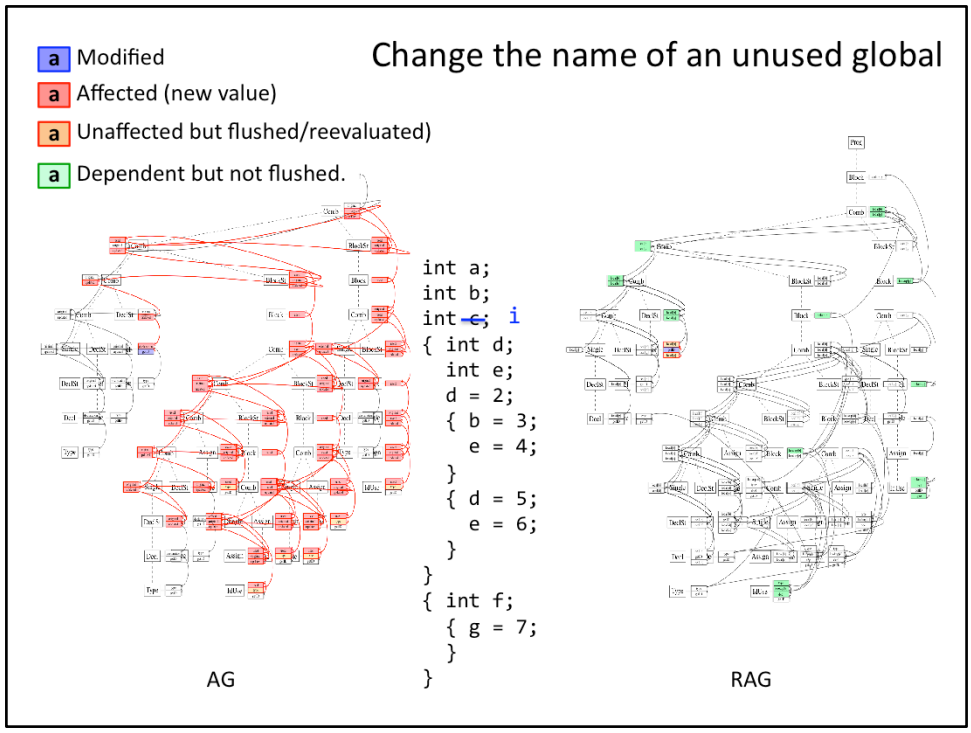
	AGs	RAGs
Symbol tables/Environments	Encoded in attributes	The attributed AST itself
Size of attribute values	Many very large environment values	Small: typically a single reference
Computation of attributes	Typically data-driven, strict evaluation	By demand, non-strict
Incremental evaluation	Optimal change propagation algorithm [Reps82]	Non-optimal algorithm. Flush dependent attributes. Abort flush for cache-independent attributes.
Number of affected attribute after AST modification	Many	Few
Performance	Non-practical, even if optimal. Does not scale.	Optimal for practical cases. Scales to large programs.

For ordinary AGs, there is an optimal change propagation algorithm (Reps 82). (The number of reevaluated attributes is O(AFFECTED).)
RAGs cannot use this algorithm, but have much fewer affected attributes.



A cache-independent attribute depends only on the tree and tokens. It does not depend on any cached attribute value.

Such attributes can be reevaluated by the incremental RAG algorithm, and propagation of flush can be aborted if the value is unchanged.



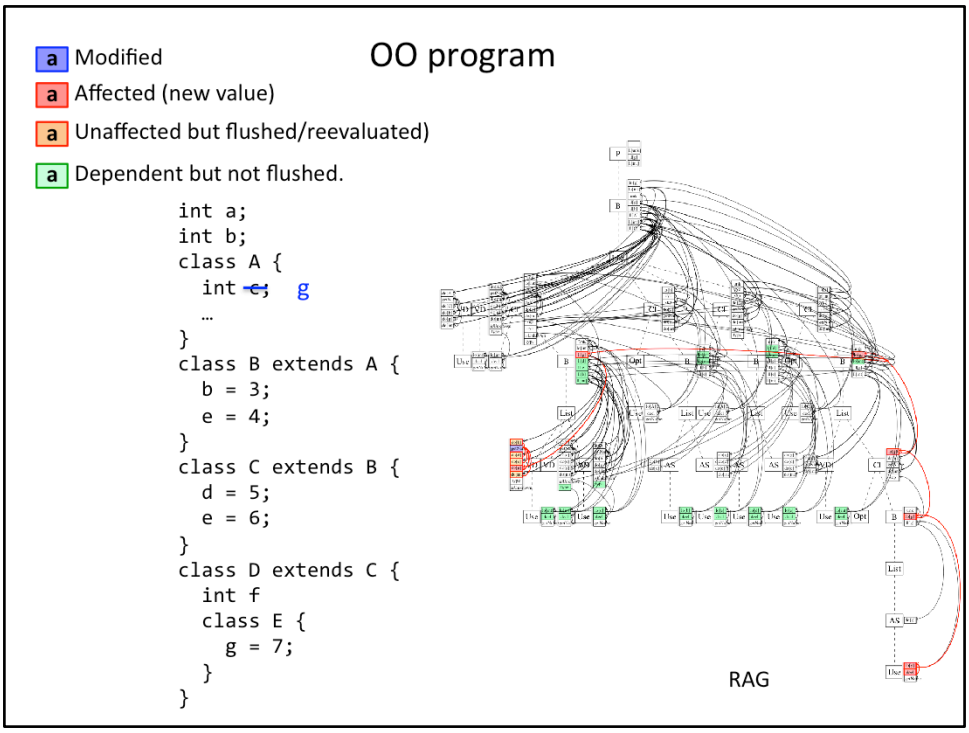
In this example we change an unused global variable c to another unused name i.
Intuitively, we would expect this change to not have any effect at all on attributes.

Here there are very many affected attributes in the AG.
The RAG actually has no affected attributes at all.

A couple of cache-independent attributes are reevaluated for the RAG (the orange ones).
But since the values of those attributes are the same, propagation of flush is aborted at this point.

The green attributes are those that *could* change value, depending on the actual change to the declaration c.
For example, if we changed ”c” to ”g”, then some of the green attributes would need to be reevaluated.

So if we had a naive flush propagation (without checking cache-independent attributes), we would have flushed also the green attributes.



Here is an example for an object-oriented program.
When the unused variable c is renamed to g, and g is used inside the class E.

The affected attributes here are the decl attribute of the g-use, lookup(”g”) attributes in blocks visible from the g-use, and the localLookup(”g”) of the block containing the modified declaration.

Status

- Fine- and coarse-grained dependency tracking and flushing
- Dynamic recognition of cache-independent attributes
- Abortion of flush propagation for unchanged cache-independent attributes
- Ongoing implementation of ”fast” local lookups (complete evaluation at first access)
- Ongoing refactorization of JastAddJ to only use ”small” attributes for critical parts of analysis.
- Ongoing evaluation
- Tech report available

The tech report: E. Söderberg and G. Hedin: Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking. April 2012. http://fileadmin.cs.lth.se/sde/publications/reports/2012-Soderberg-RAGsIncEval-report.pdf

Conclusions

- Parameterized attributes allows computed properties to be split into very small pieces.
- This is a key to enabling good incremental behavior.