

Reference Attribute Grammar controlled Rewriting

Motivation and Overview

Christoff Bürger

Department of Computer Science, Faculty of Engineering, LTH

Lund University

Lund, Sweden

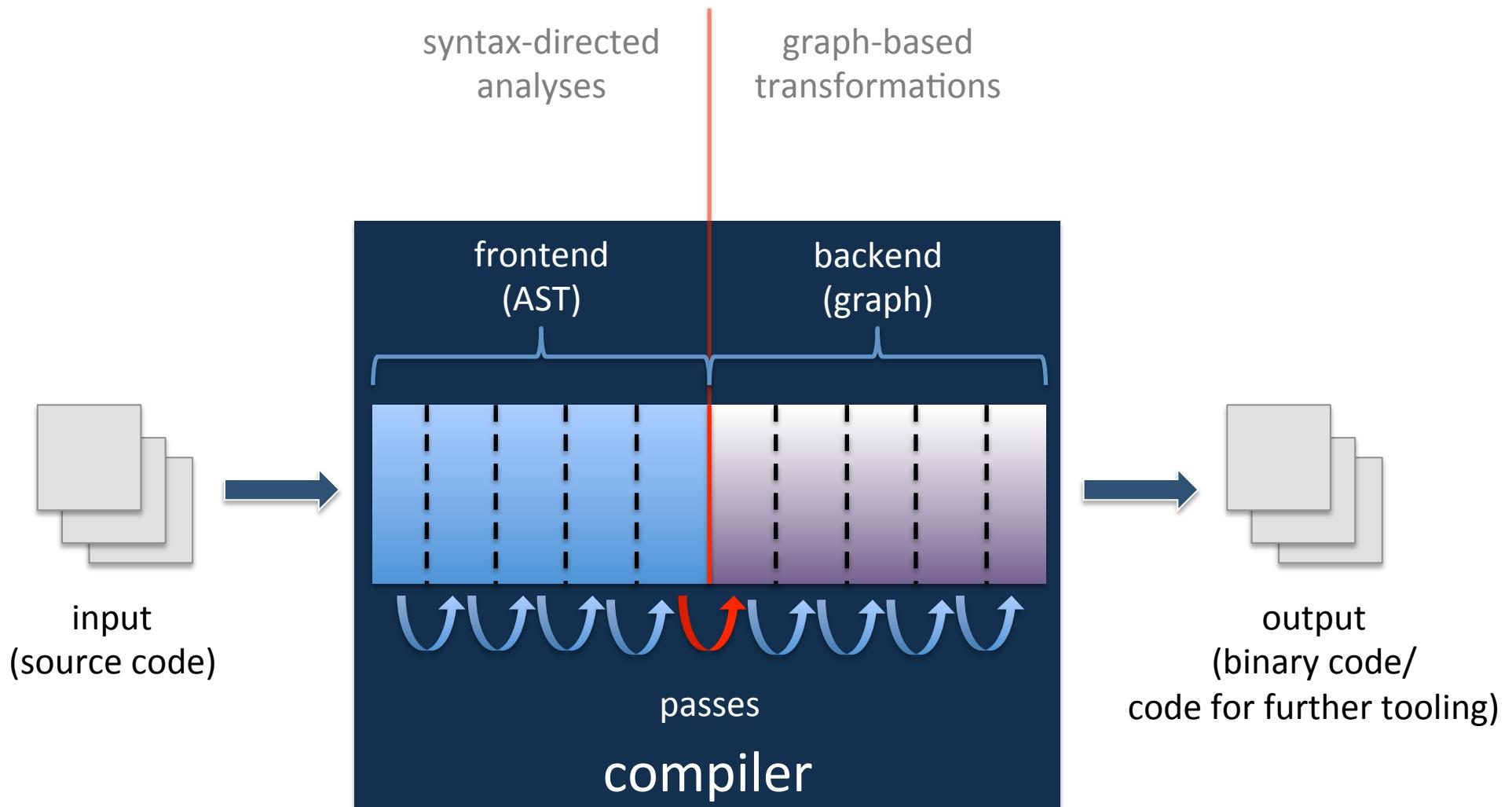
christoff.burger@cs.lth.se

The Problem

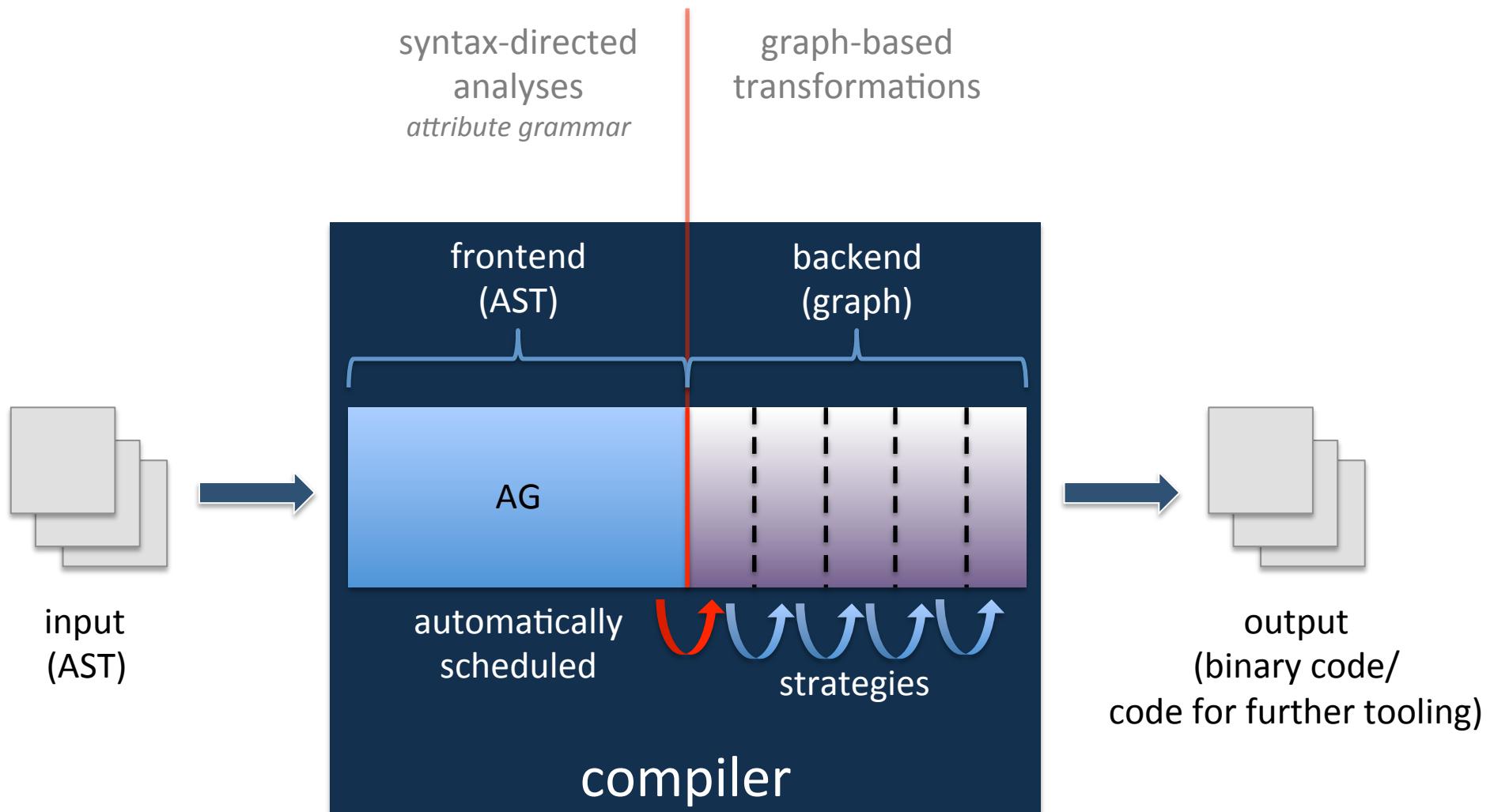
What do you want to achieve?

Interactive, mutual-dependent analyses
& transformations

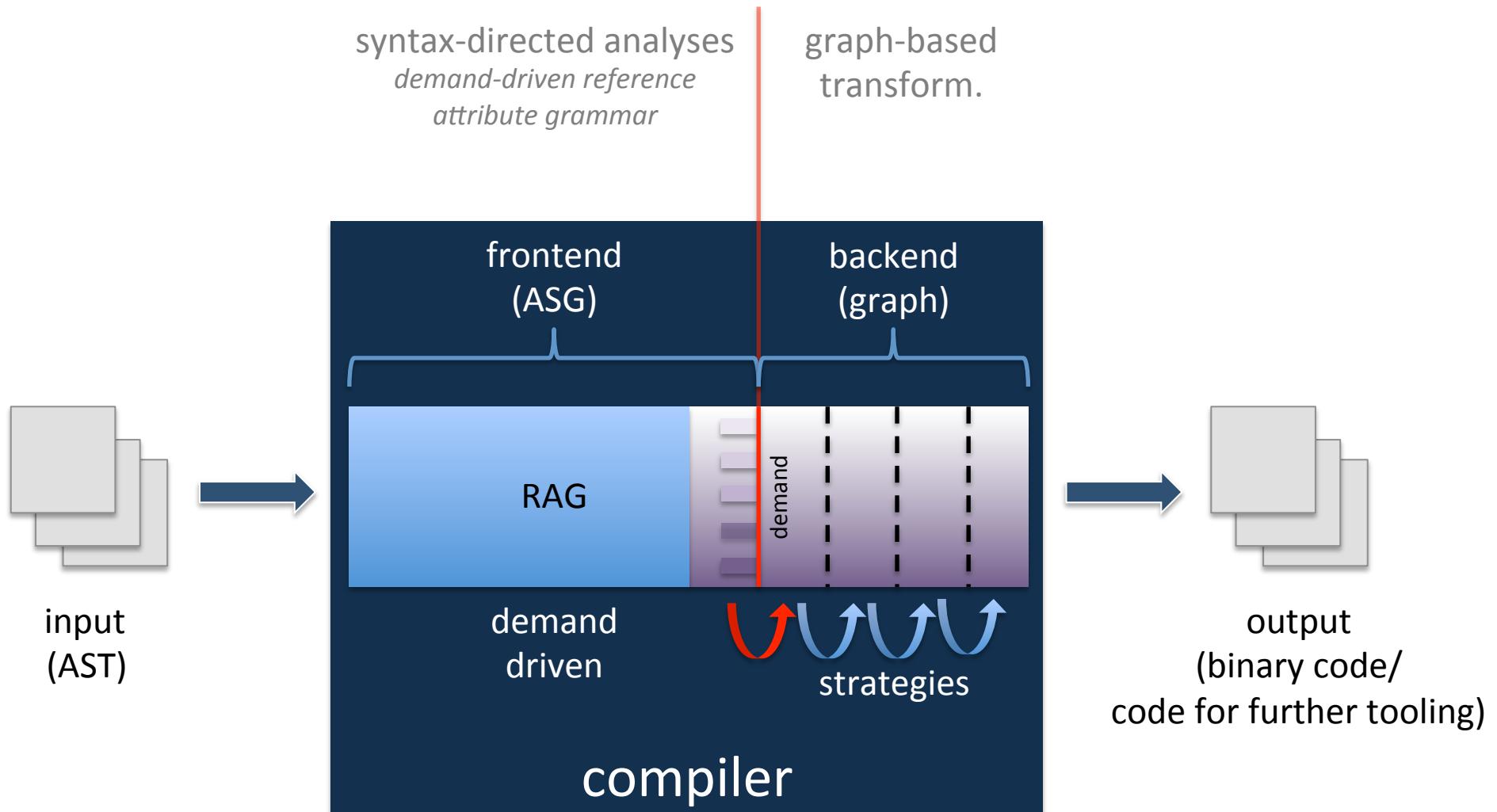
From batch to interactive



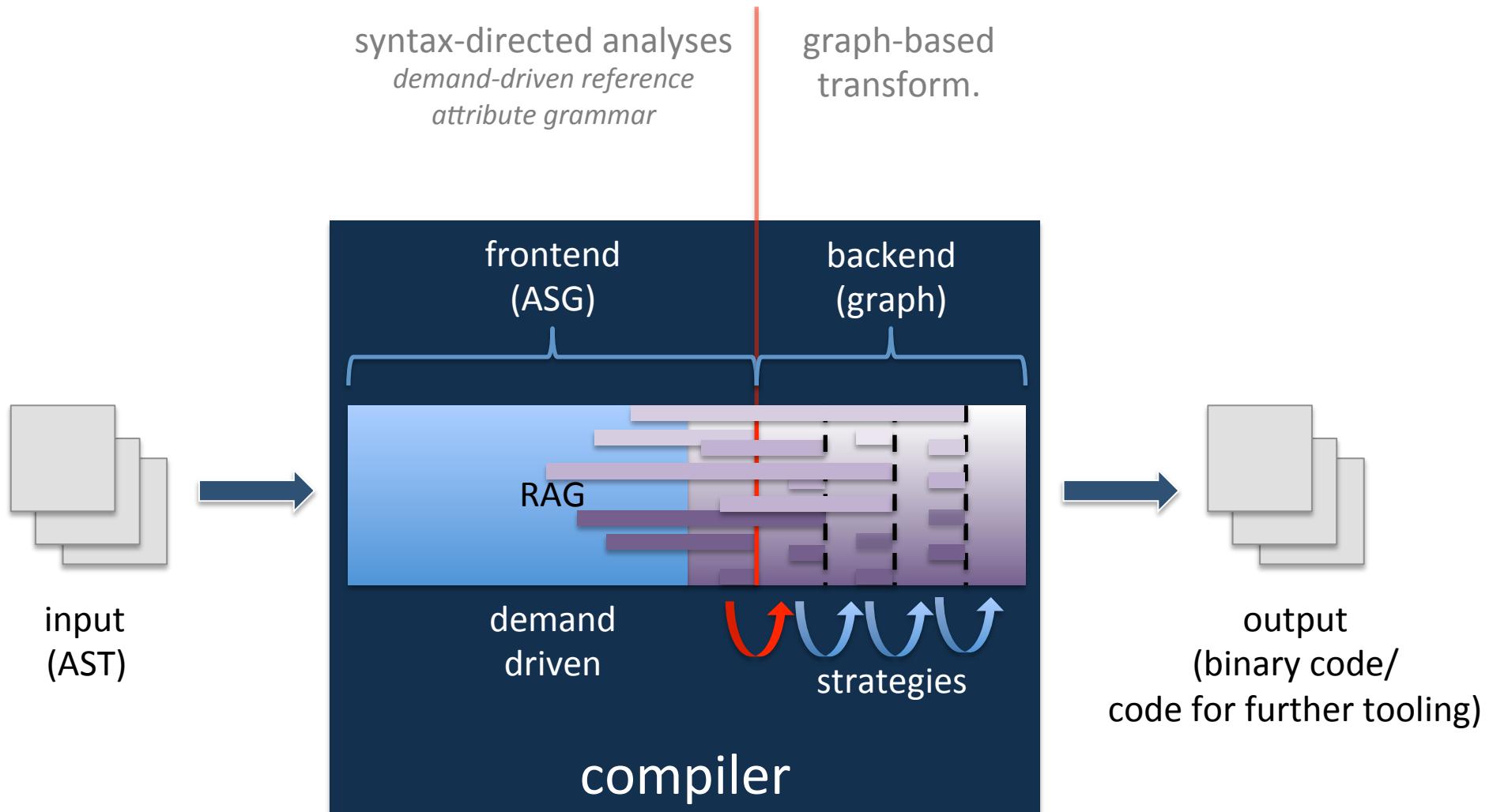
From batch to interactive



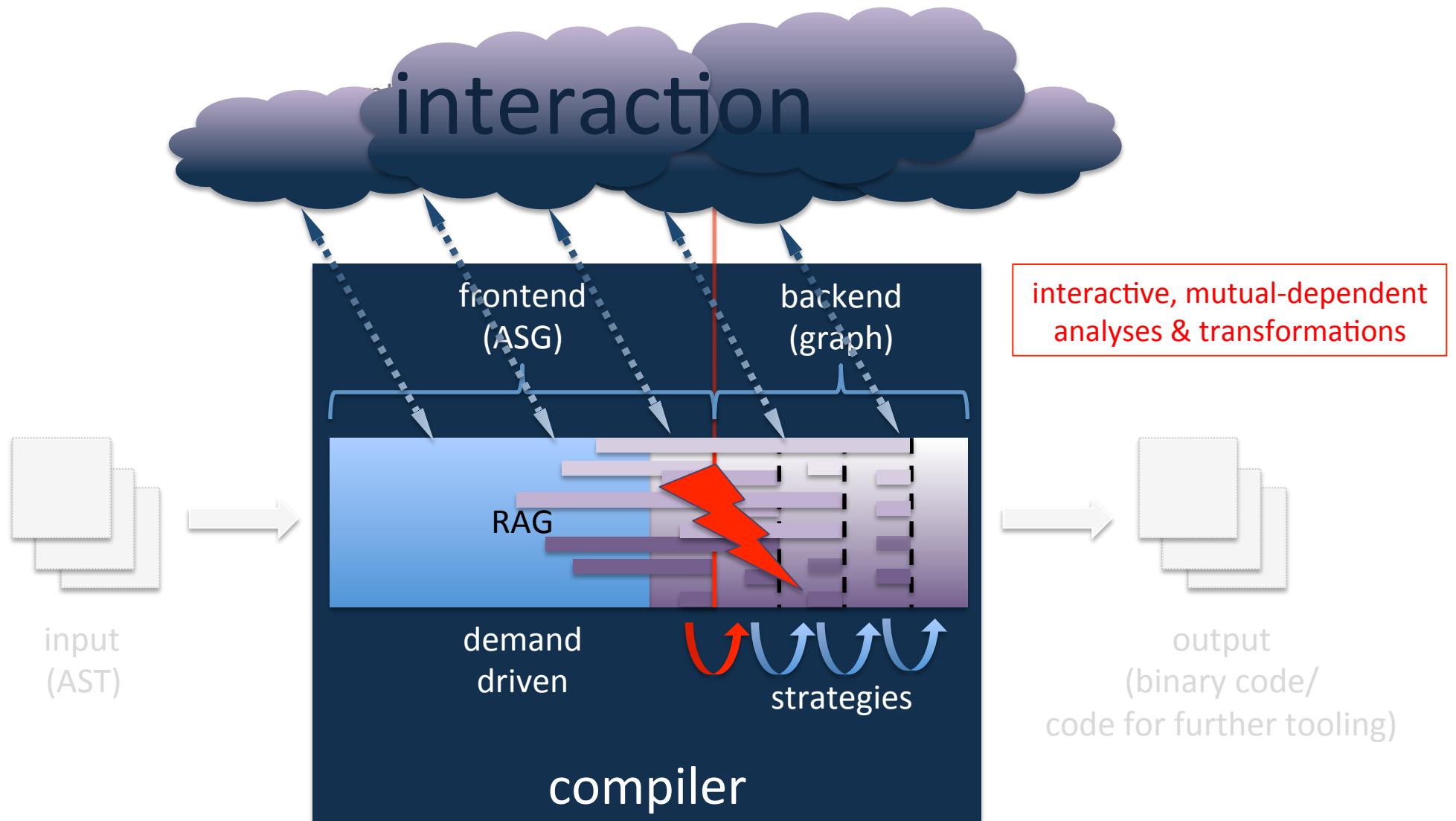
From batch to interactive



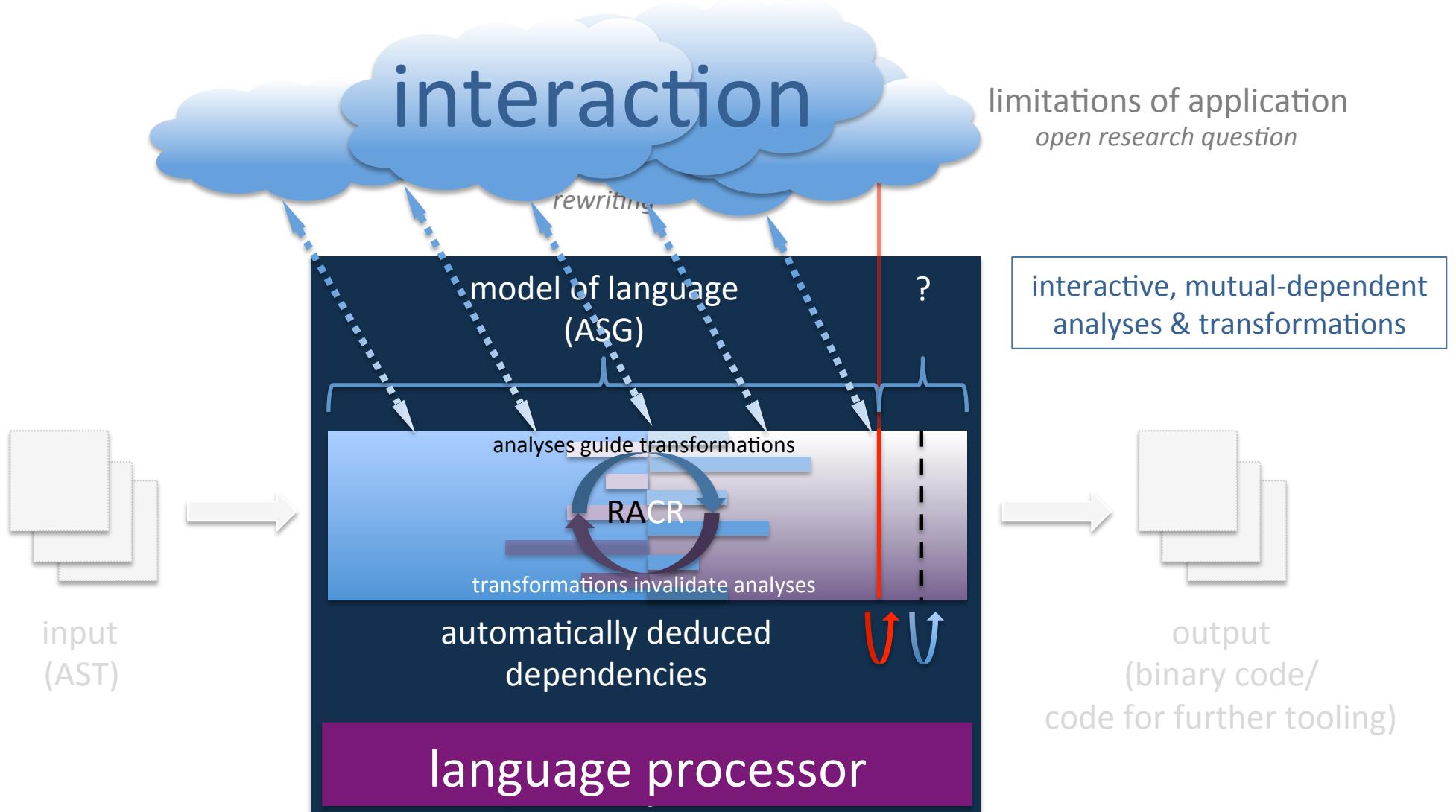
From batch to interactive



From batch to interactive



From batch to interactive



The Solution

What is RAG-controlled rewriting?

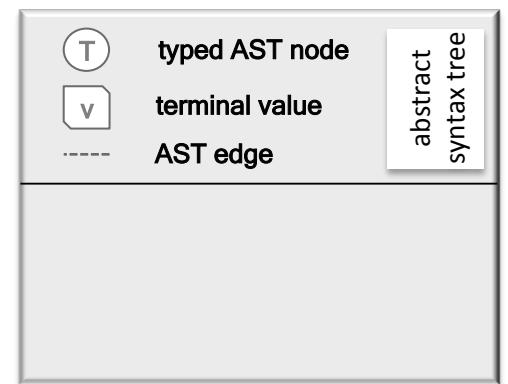
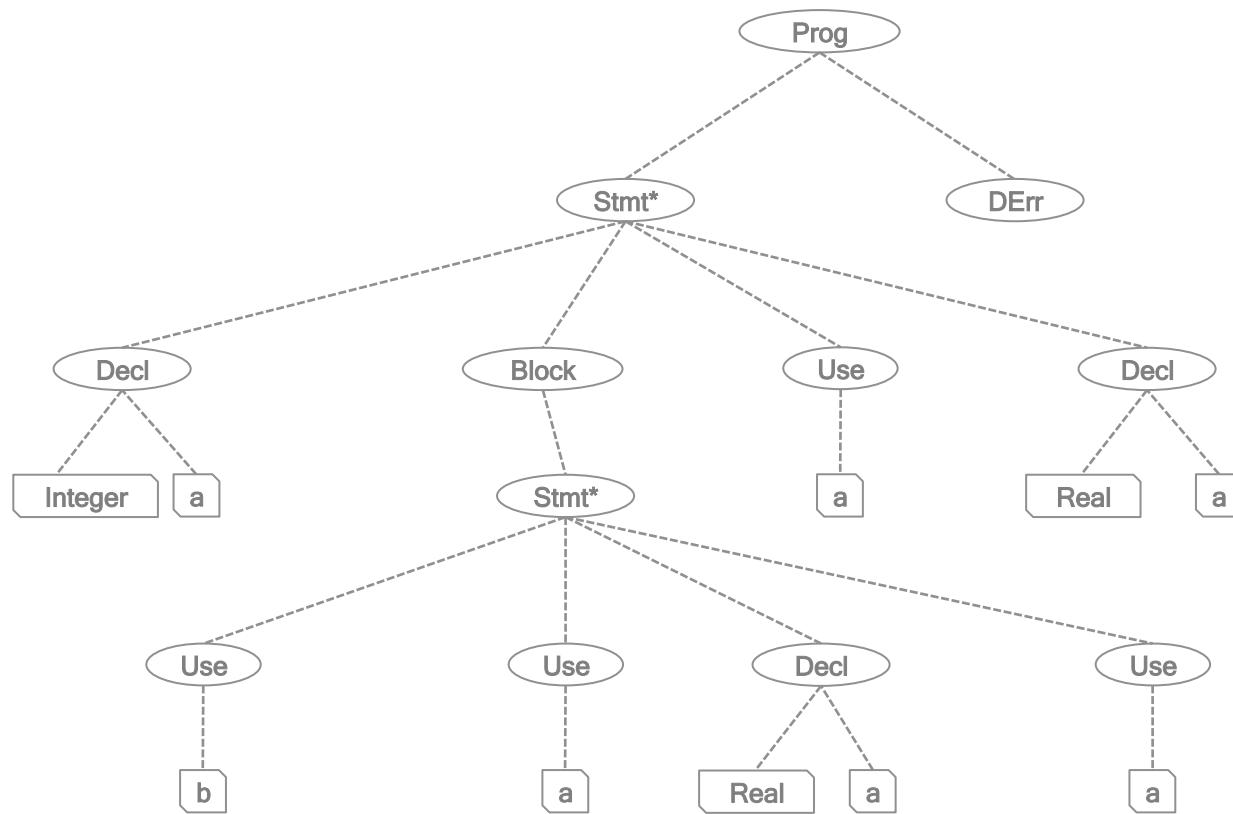
Reference attribute grammars, ASGs,
RAG-controlled rewriting & *RACR*¹

¹ <https://github.com/christoff-buerger/racr>

Reference attribute grammars & ASGs

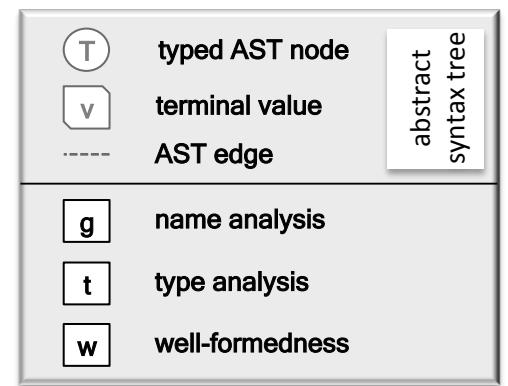
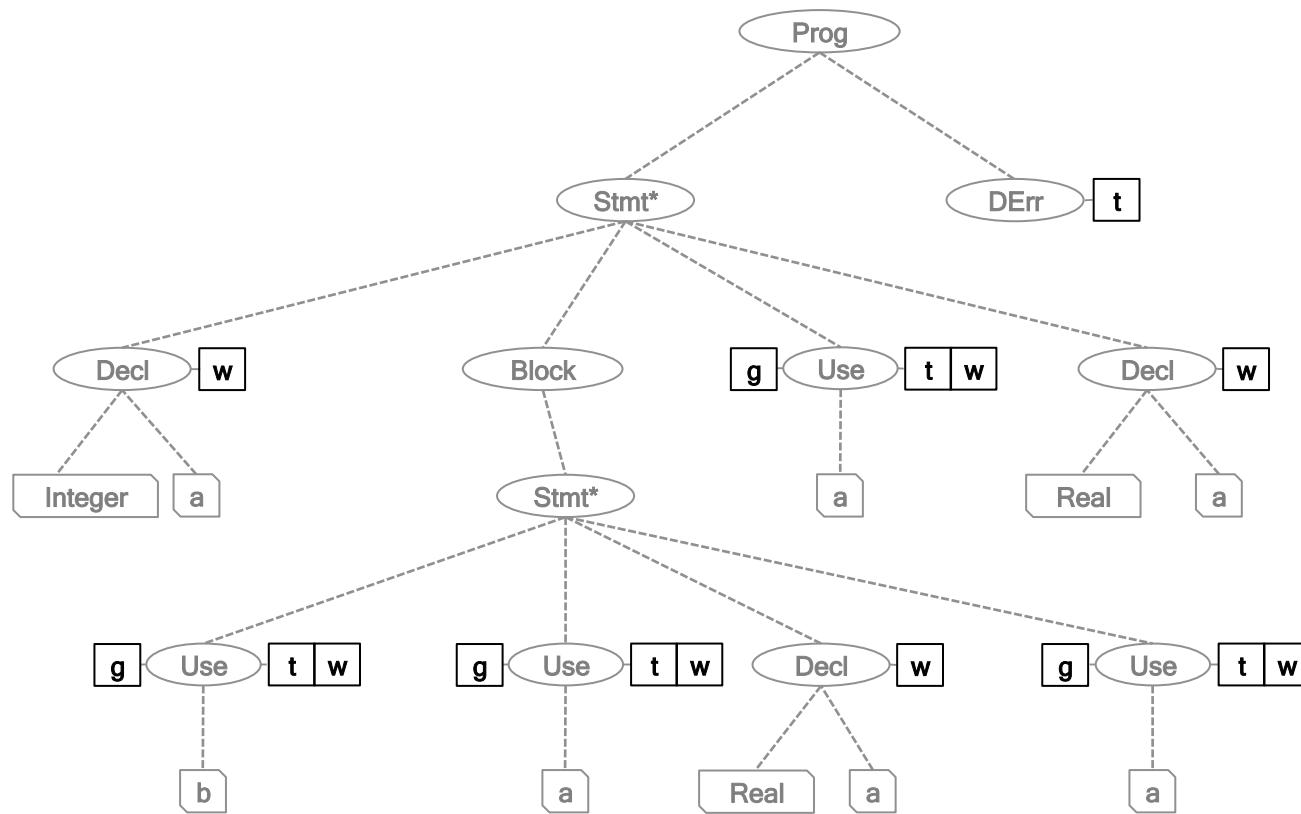
```
Program
  decl a : integer
  Begin
    use b ; Error
    use a
    decl a : real
    use a
  End
  use a
  decl a : real ; Error
End
```

Reference attribute grammars & ASGs



Program
decl a : integer
Begin
use b ; Error
use a
decl a : real
use a
End
use a
decl a : real ; Error
End

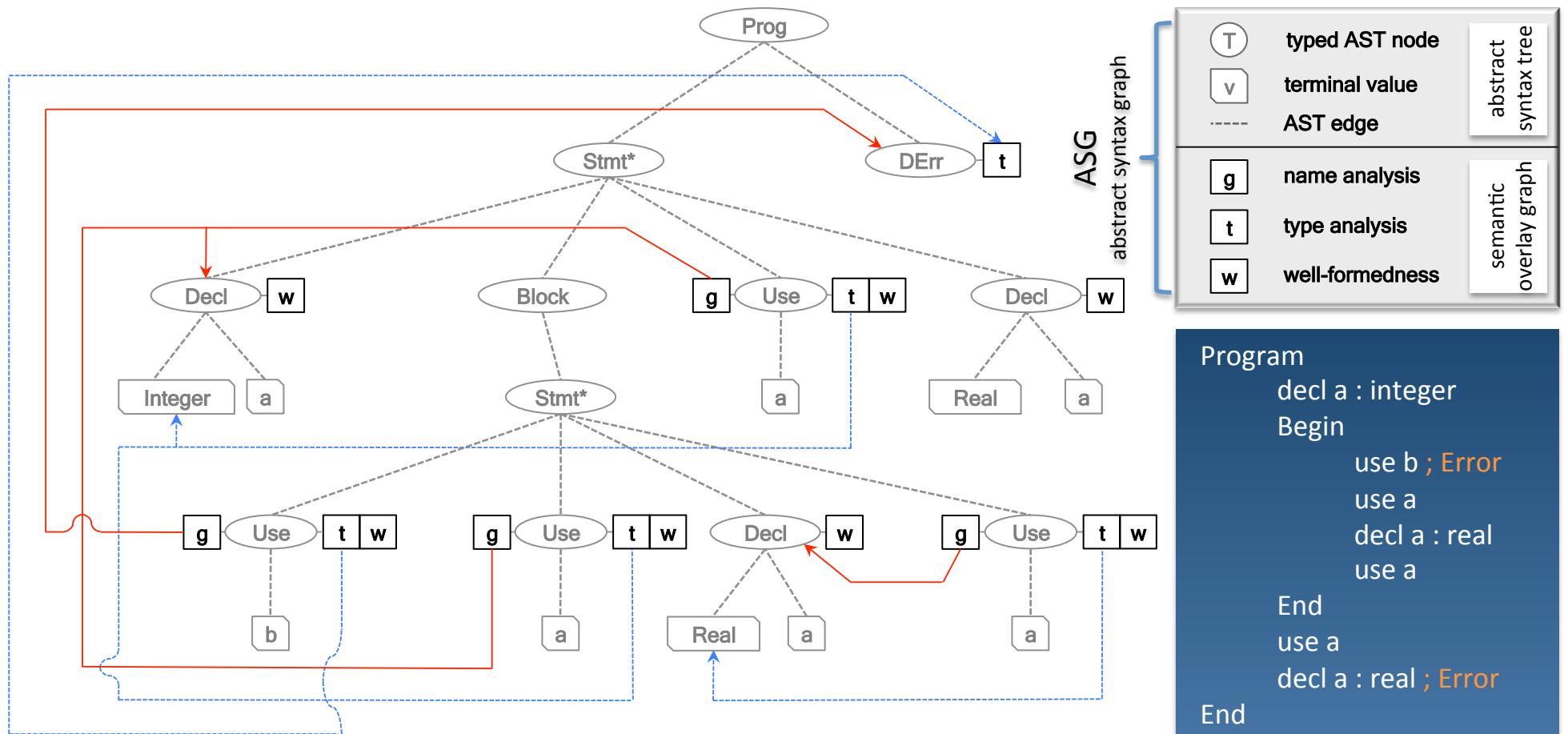
Reference attribute grammars & ASGs



```

Program
  decl a : integer
  Begin
    use b ; Error
    use a
    decl a : real
    use a
  End
  use a
  decl a : real ; Error
End
  
```

Reference attribute grammars & ASGs



RAG-controlled rewriting

- RAG-controlled rewriting = **RAGs** + **rewriting**
 - RAG for declarative **analyses**
 - graph rewriting for ASG **transformations**
 - seamless combination:
 - use of analyses to deduce rewrites
 - rewrites automatically update analyses
- >> incremental
- 
- mutual control

RACR

Reference implementation of RAG-controlled
rewriting in *Scheme*.

<https://github.com/christoff-buerger/racr>

The Implementation

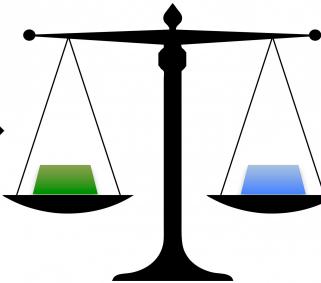
How works RAG-controlled rewriting?

Dynamic attribute dependency graphs &
incremental evaluation

Query & rewrite functions

well balanced set of functions

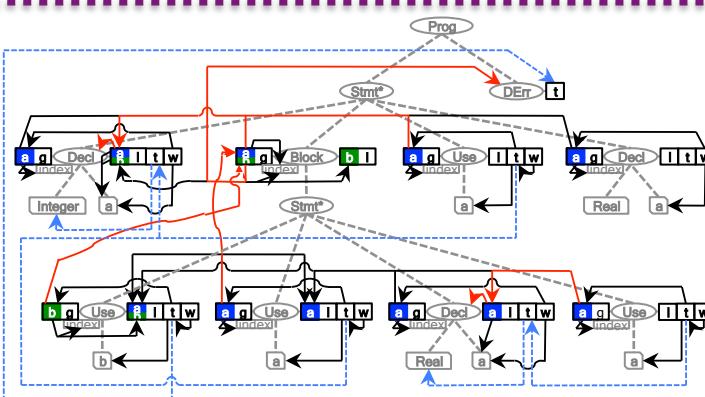
query functions to access ASG
add dependencies on queried information



rewrite functions to change ASG
invalidate attributes depending on rewritten information

query functions

- only way to query ASG
- used in attribute equations



construct

dynamic attribute
dependency graph

rewrite functions

- only way to change ASG
- used for transformations

invalidate

demand-driven evaluation avoids unnecessary computations

Query & rewrite functions

query functions

$(=Name n . a)$	value of attribute <i>Name</i>
$(->c n)$	child <i>c</i> of <i>n</i> (<i>c</i> can be index)
$(<- n)$	parent of <i>n</i>
$(->c? n)$	has <i>n</i> a <i>c</i> child (<i>c</i> can be index)
$(<-? n)$	has <i>n</i> a parent
$(index n)$	child-position of <i>n</i>
$(num-children n)$	number of children of <i>n</i>
$(T=? n)$	is <i>n</i> exactly of type <i>T</i>
$(T<? n)$	is <i>n</i> subtype of type <i>T</i>
$(T>? n)$	is <i>n</i> supertype of type <i>T</i>
$(\{=,<,>\}? n1 n2)$	is <i>n1</i> $\{\!=,\,<,>\}$ -type of <i>n2</i>
$(find f n . b)$	find child of <i>n</i> satisfying <i>f</i>

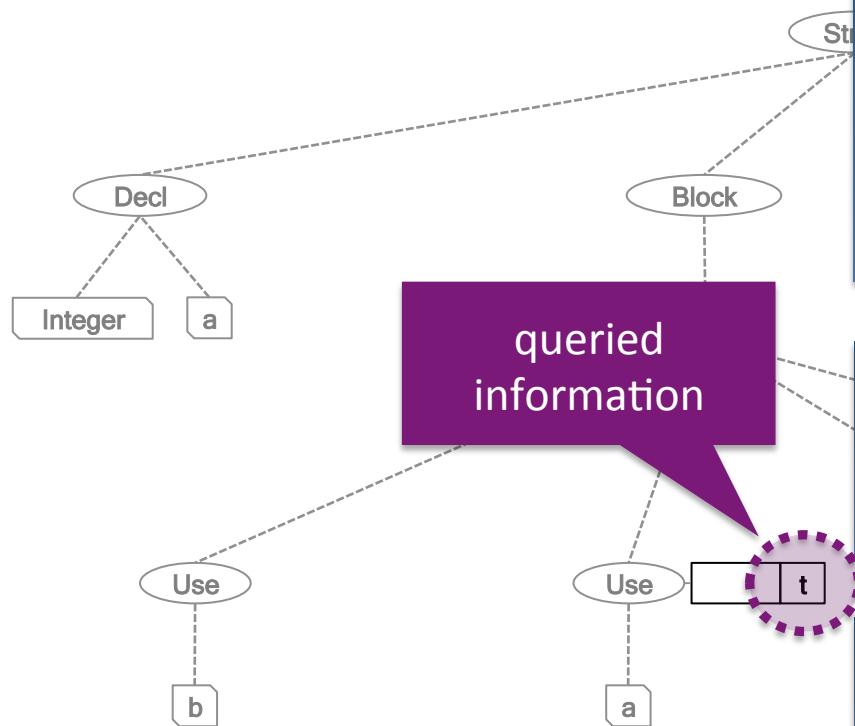
rewrite functions

$(r-subtree n1 n2)$	replace <i>n1</i> by <i>n2</i>
$(r-terminal t v)$	replace value of terminal <i>t</i>
$(add n l)$	add <i>n</i> to list <i>l</i>
$(insert n i l)$	insert <i>n</i> at position <i>i</i> in <i>l</i>
$(delete n)$	delete list element <i>n</i>
$(refine n T . c)$	refine <i>n</i> to subtype <i>T</i>
$(abstract n T)$	abstract <i>n</i> to supertype <i>T</i>

dependency types

value, exists, has-child(*child/index*), has-parent, index, num-children, type, subtype(*T*), supertype(*T*)

Dynamic attribute dependency graphs

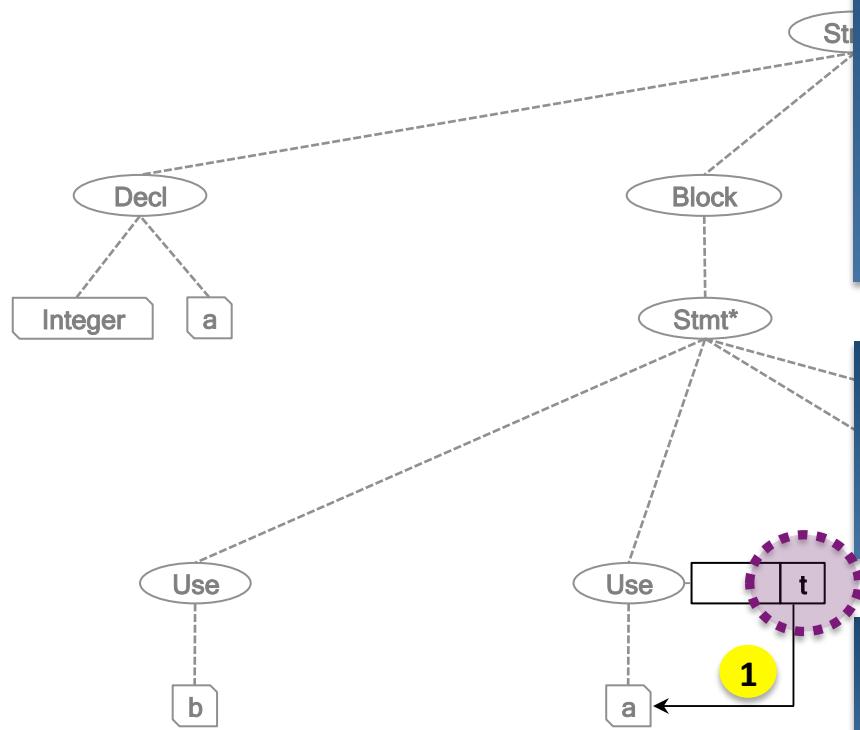


```
(ag-rule G-Decl ; Inherited attribute  
((Block Stmt*) ; Equation for statements of blocks  
 (lambda (n name)  
 (or (find-L-Decl name (<- n) (index n))  
 (=G-Decl (<- (<- n)) name))))  
((Prog Stmt*) ; Equation for statements of programs  
 (lambda (n name)  
 (or (find-L-Decl name (<- n) (index n))  
 (->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute  
(Stmt (lambda (n name) #f))  
(Decl (lambda (n name)  
 (if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute  
(Use (lambda (n) (=Type (=G-Decl n (->name n))))))  
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



```

(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
  
```

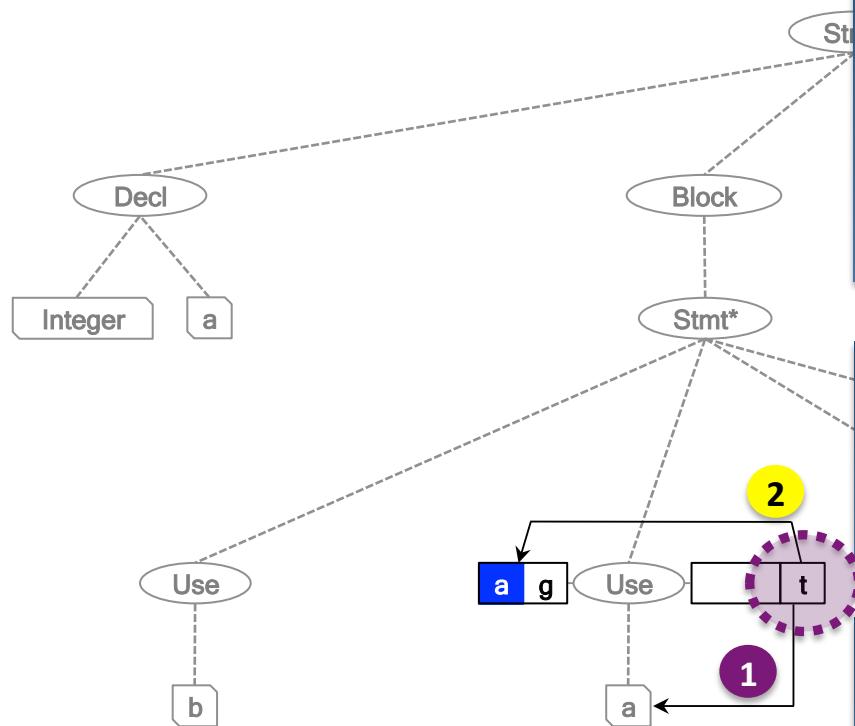
```

(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
  
```

```

(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
  
```

Dynamic attribute dependency graphs

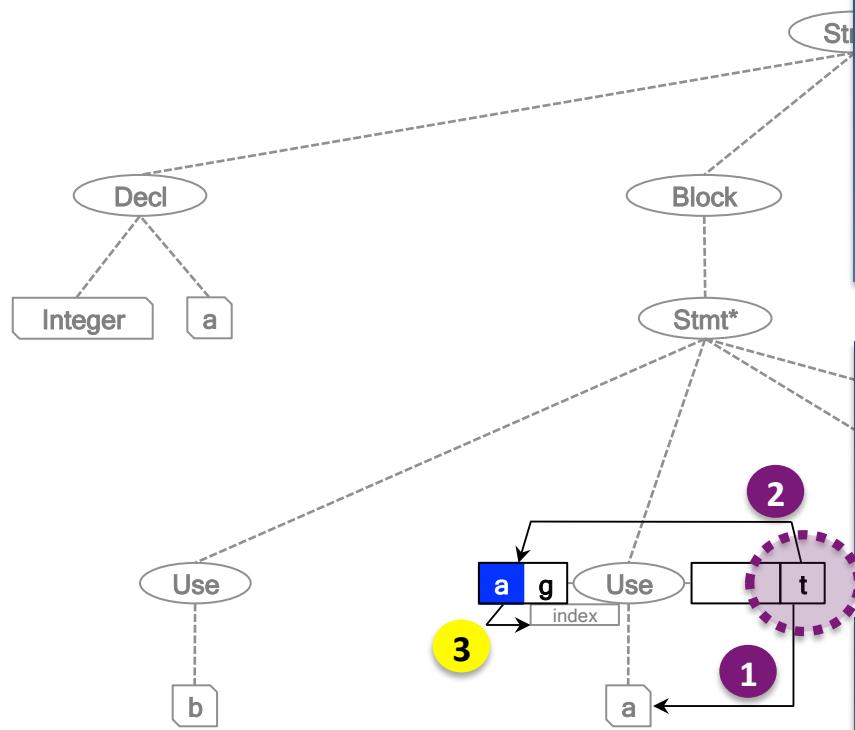


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
 (lambda (n name)
  (or (find-L-Decl name (<- n) (index n))
      (=G-Decl (<- (<- n)) name))))
 ((Prog Stmt*) ; Equation for statements of programs
 (lambda (n name)
  (or (find-L-Decl name (<- n) (index n))
      (->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
 (lambda (n name)
 (or (find-L-Decl name (<- n) (index n))
 (=G-Decl (<- (<- n)) name))))
 ((Prog Stmt*) ; Equation for statements of programs
 (lambda (n name)
 (or (find-L-Decl name (<- n) (index n))
 (->DErr (<- (<- n)))))))
```

a

Real

a

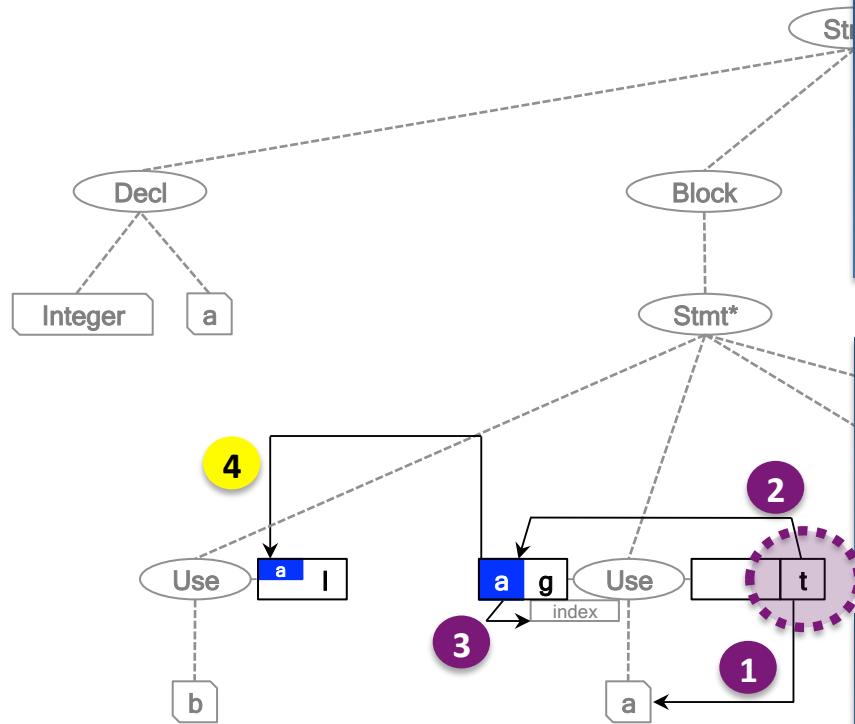
```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

Decl

Use

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

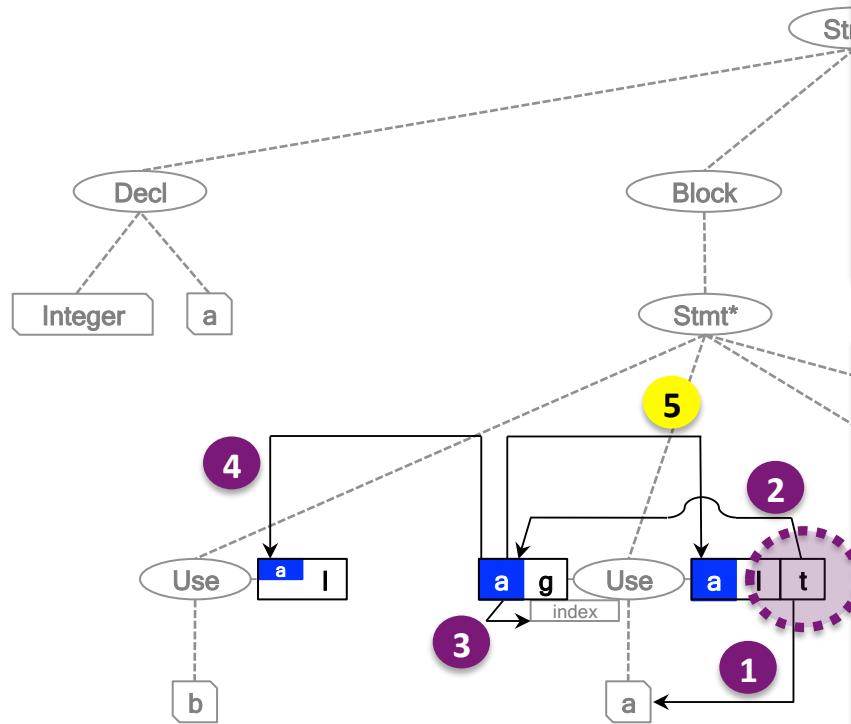


```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```



```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
 (lambda (n name)
 (or (find-L-Decl name (<- n) (index n))
 (=G-Decl (<- (<- n)) name))))
 ((Prog Stmt*) ; Equation for statements of programs
 (lambda (n name)
 (or (find-L-Decl name (<- n) (index n))
 (->DErr (<- (<- n)))))))
```

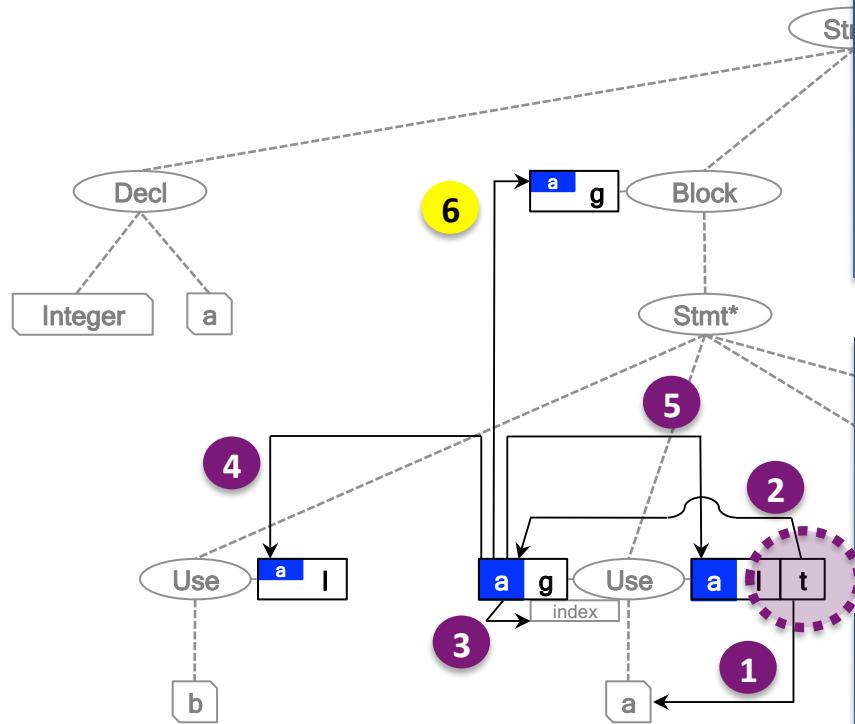


```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```



```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

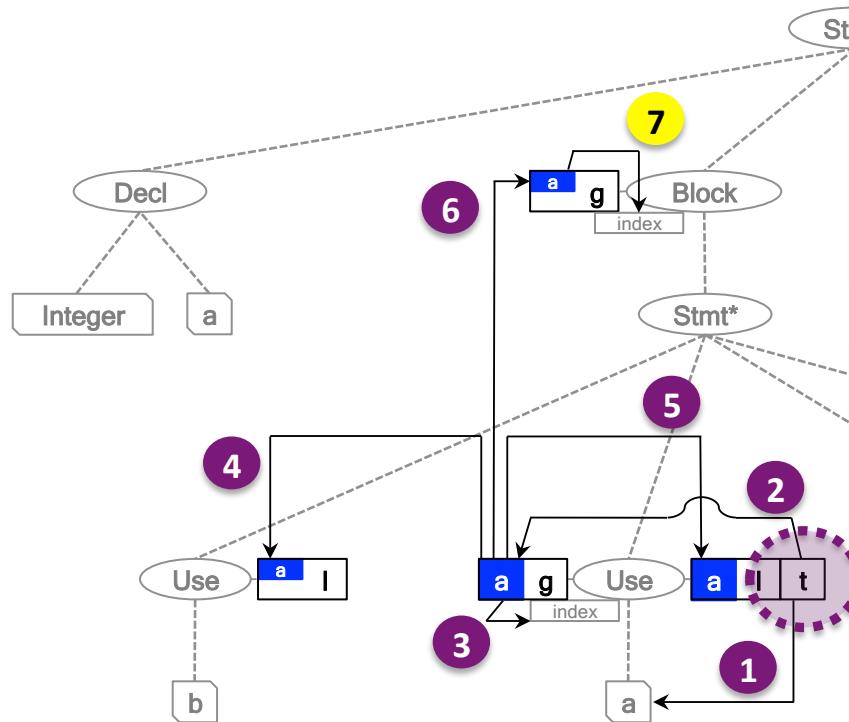


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

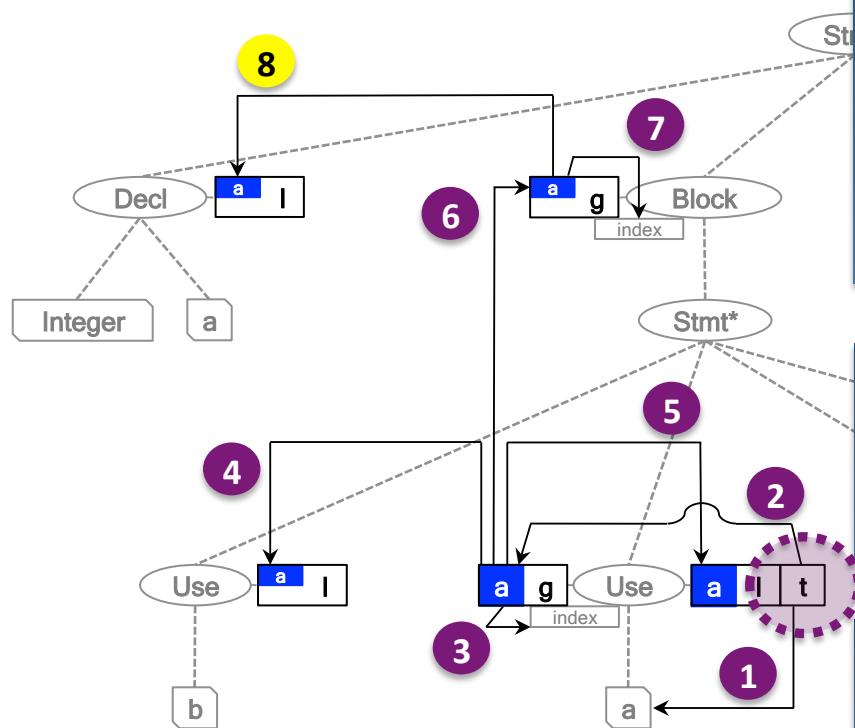


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

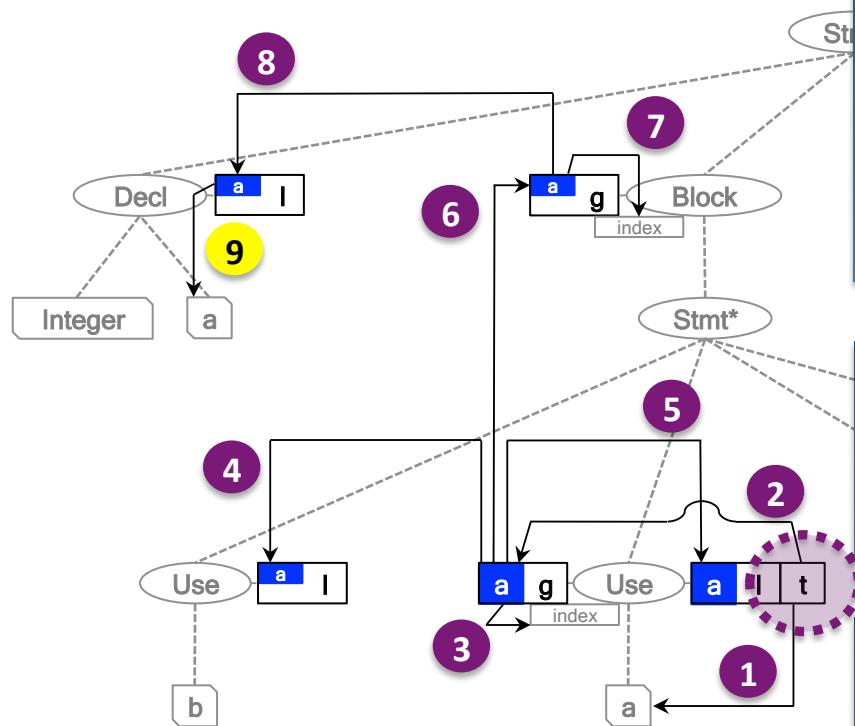


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
Decl (lambda (n name))
(if (=? (->name n) name) n #f)))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

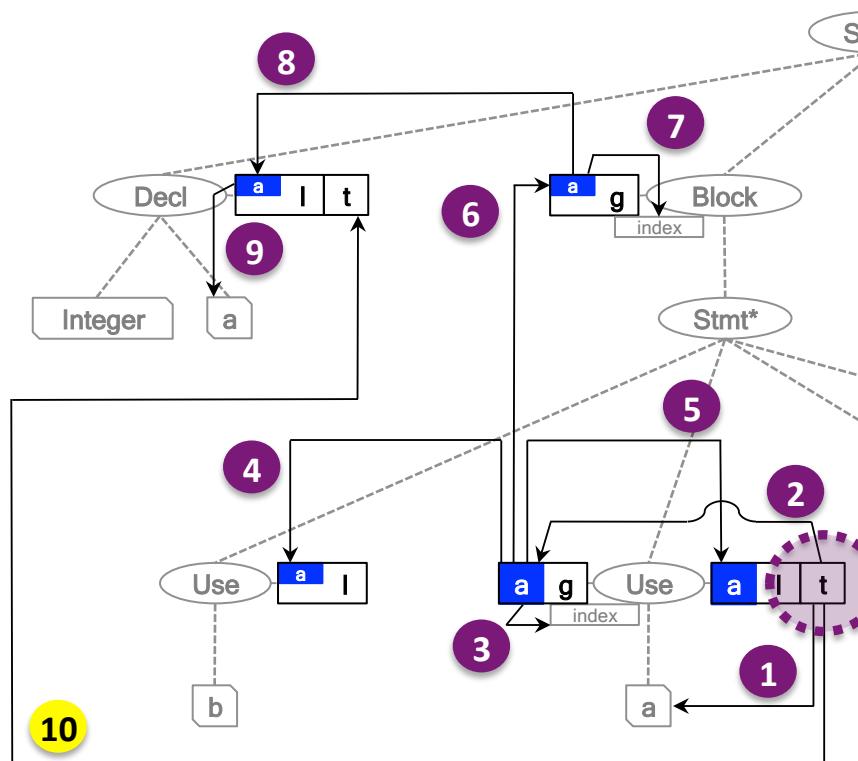


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

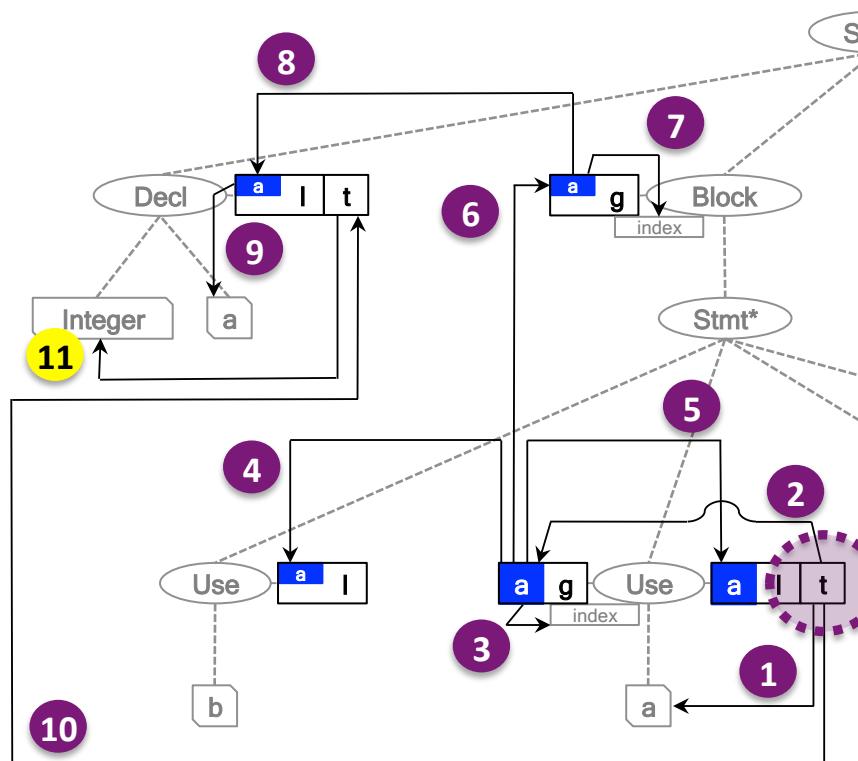


```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(=G-Decl (<- (<- n)) name))))
((Prog Stmt*) ; Equation for statements of programs
(lambda (n name)
(or (find-L-Decl name (<- n) (index n))
(->DErr (<- (<- n)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



```
(ag-rule G-Decl ; Inherited attribute
((Block Stmt*) ; Equation for statements of blocks
 (lambda (n name)
  (or (find-L-Decl name (<- n) (index n))
      (=G-Decl (<- (<- n)) name))))
 ((Prog Stmt*) ; Equation for statements of programs
 (lambda (n name)
  (or (find-L-Decl name (<- n) (index n))
      (=G-Decl (<- (<- n)) name)))))))
```

```
(ag-rule L-Decl ; Synthesised attribute
(Stmt (lambda (n name) #f))
(Decl (lambda (n name)
(if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
(Use (lambda (n) (=Type (=G-Decl n (->name n)))))
(Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs

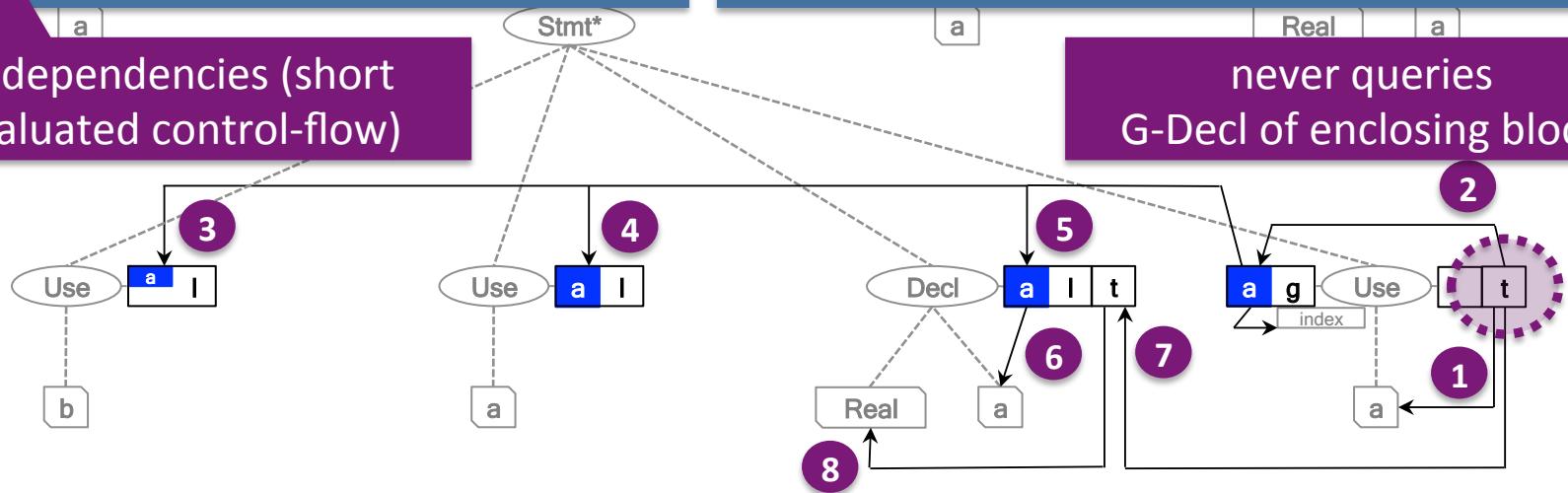
```
(ag-rule G-Decl ; Inherited attribute
  ((Block Stmt*) ; Equation for statements of blocks
    (lambda (n name)
      (or (find-L-Decl name (<- n) (index n))
          (=G-Decl (<- (<- n)) name))))
  ((Prog Stmt*) ; Equation for statements of programs
    (lambda (n name)
      (or (=L-Decl name (<- n) (index n))
          (var (<- (<- n)))))))
```

St

```
(ag-rule L-Decl ; Synthesised attribute
  (Stmt (lambda (n name) #f))
  (Decl (lambda (n name)
    (if (=? (->name n) name) n #f))))
```

```
(ag-rule Type ; Synthesised attribute
  (Use (lambda (n) (=Type (=G-Decl n (->name n)))))
  (Decl (lambda (n) (->type n))))
```

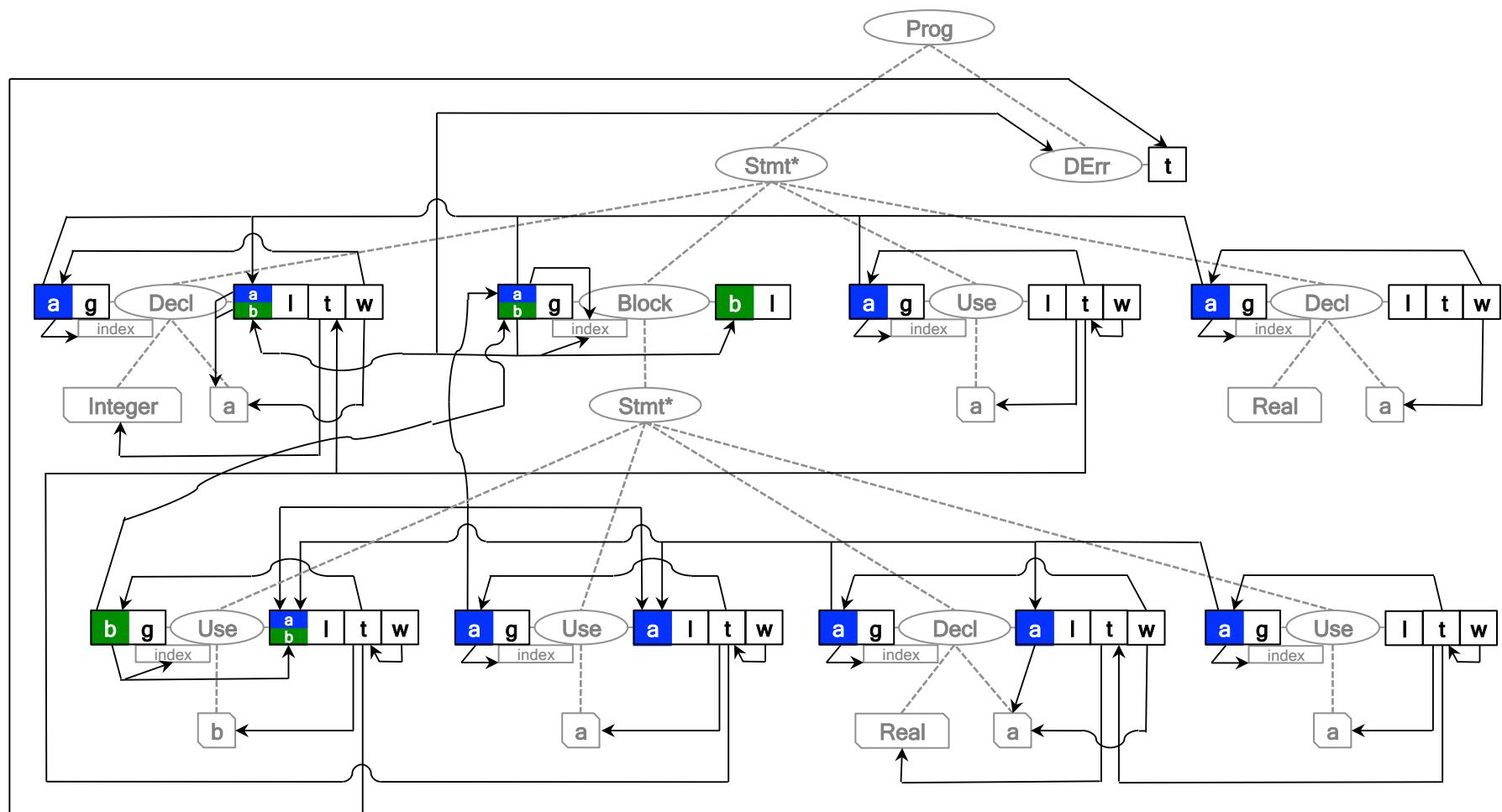
a
dynamic dependencies (short circuit evaluated control-flow)



a Real a
never queries
G-Decl of enclosing block

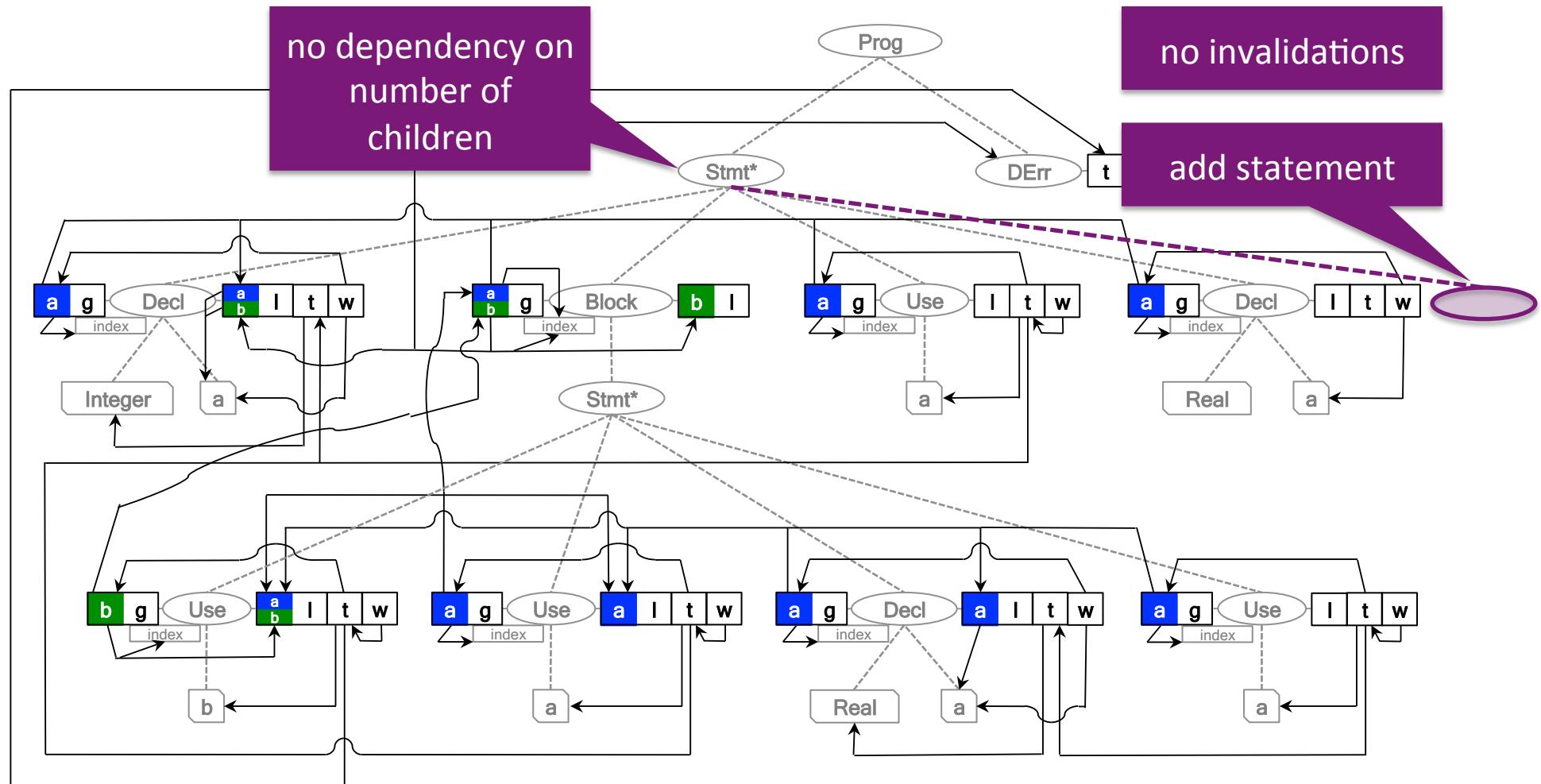
complex to achieve manually

Dynamic attribute dependency graphs

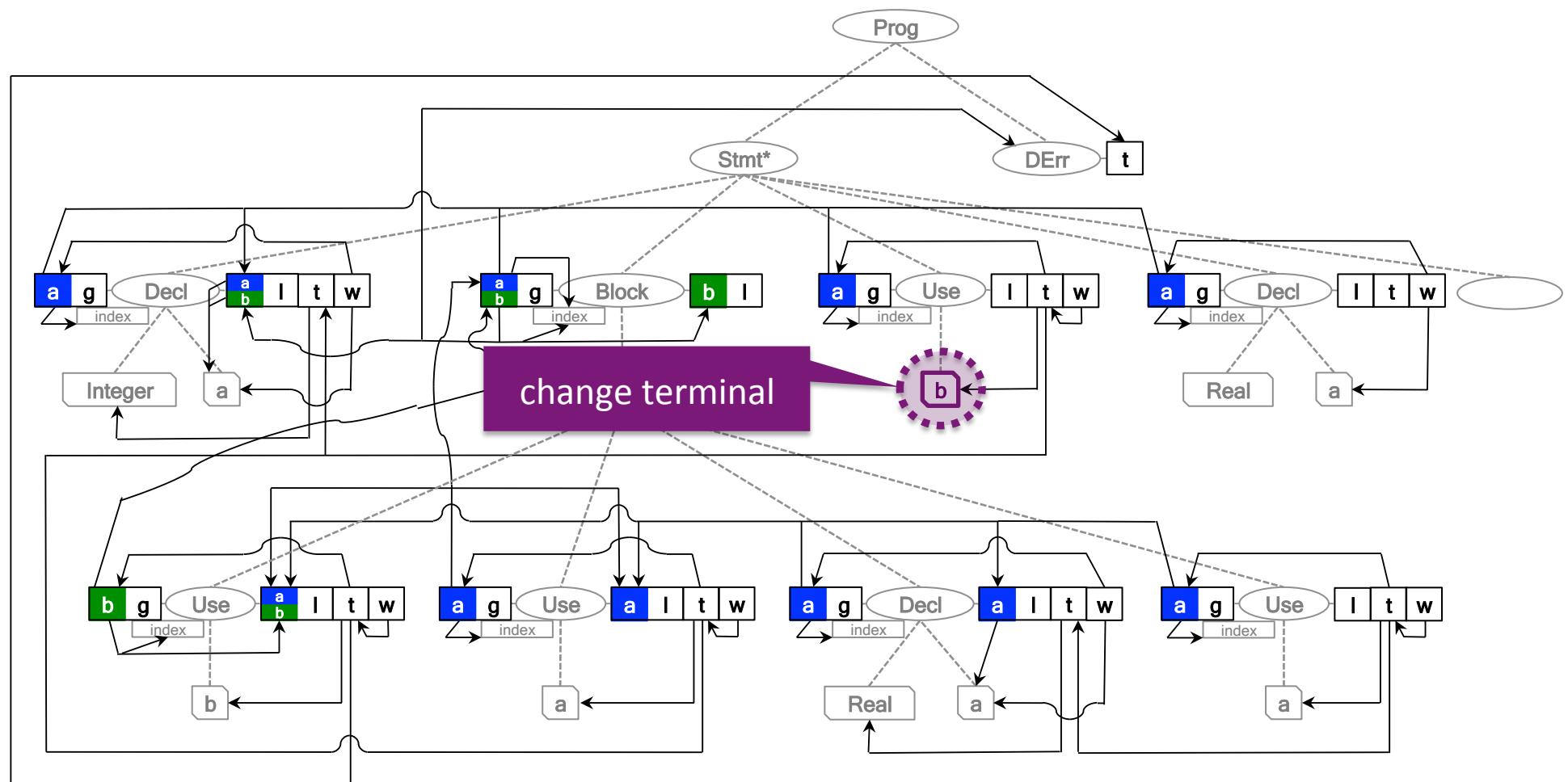


automatically in RAG-controlled rewriting

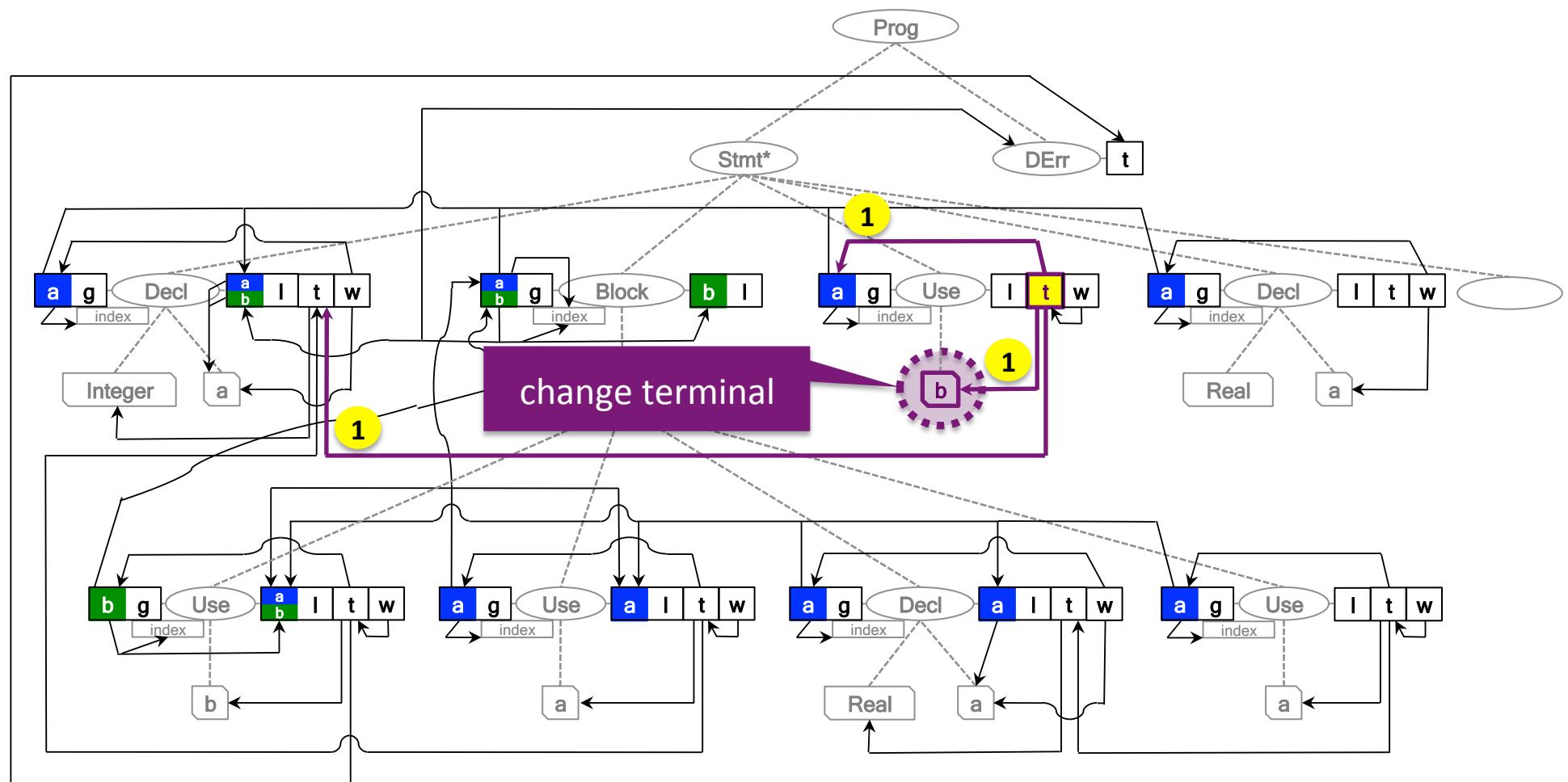
Incremental evaluation



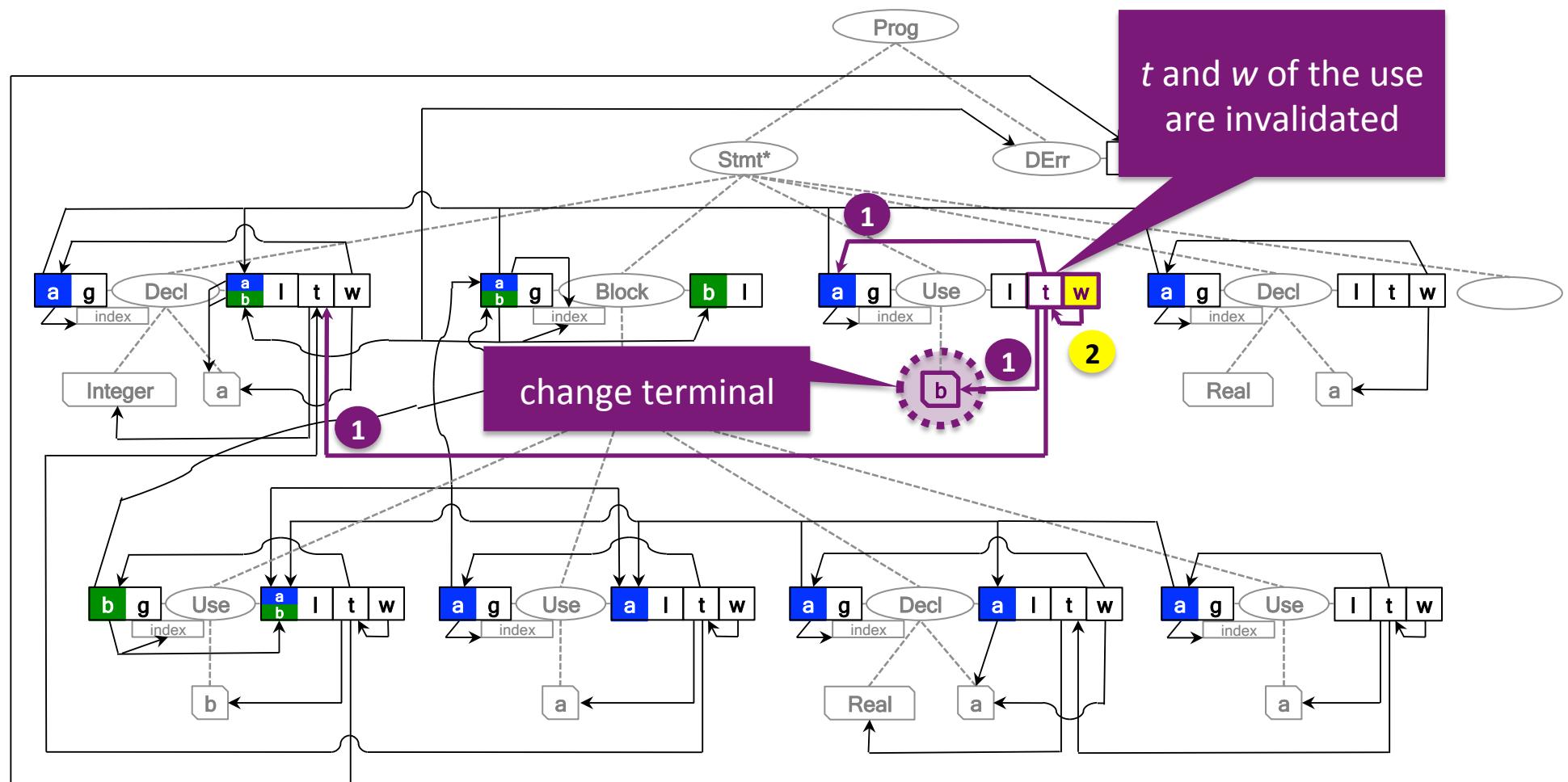
Incremental evaluation



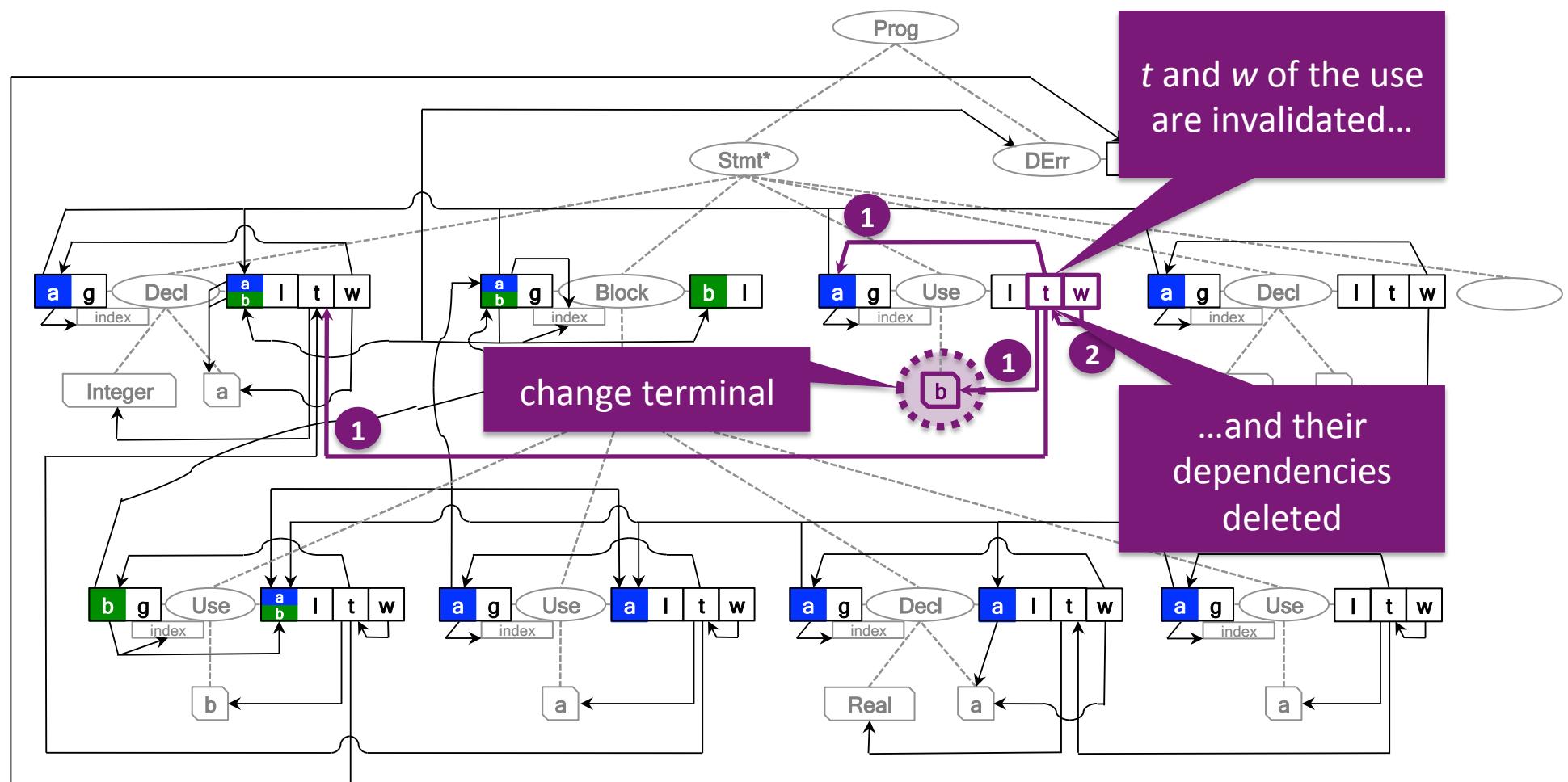
Incremental evaluation



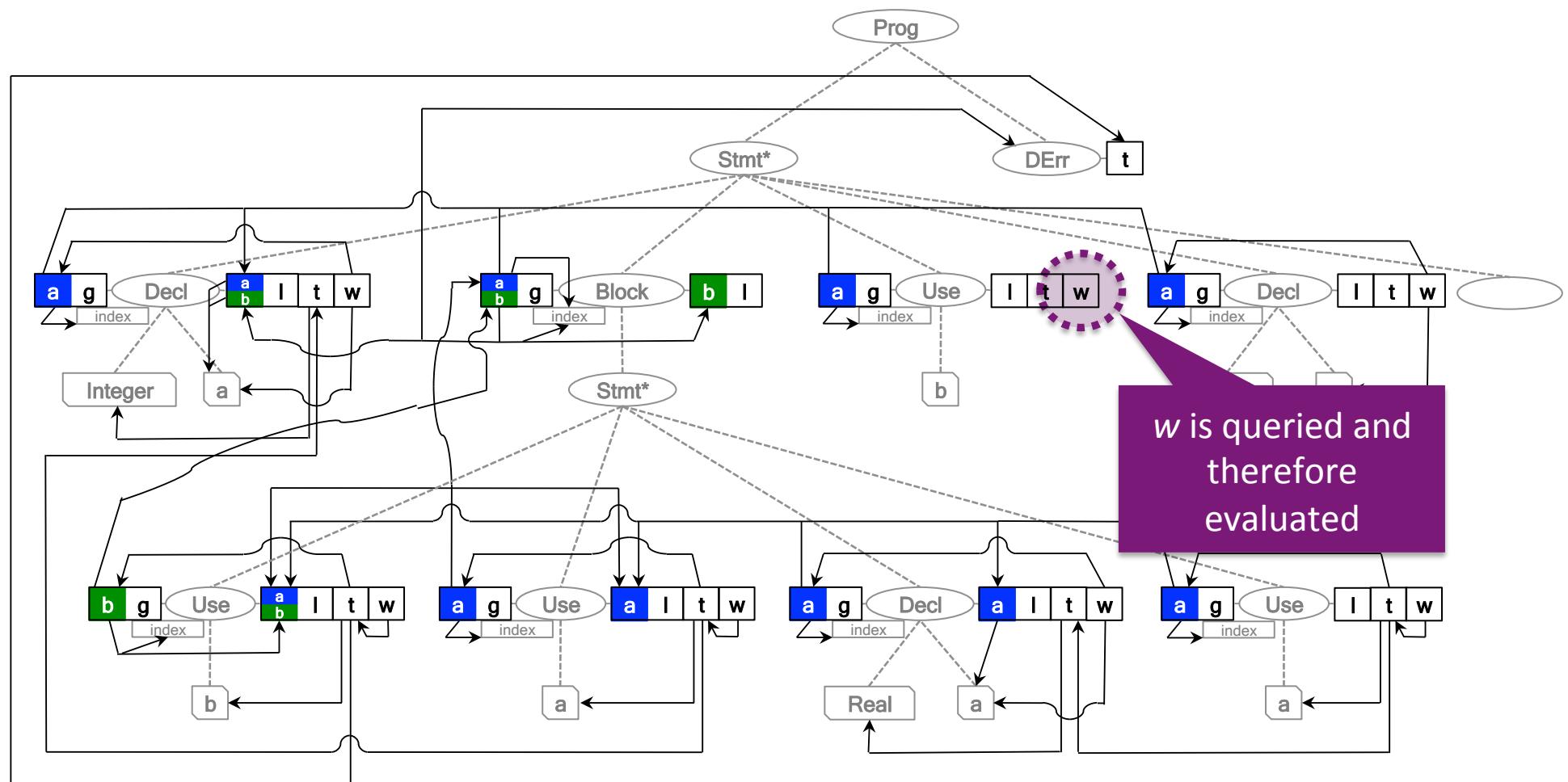
Incremental evaluation



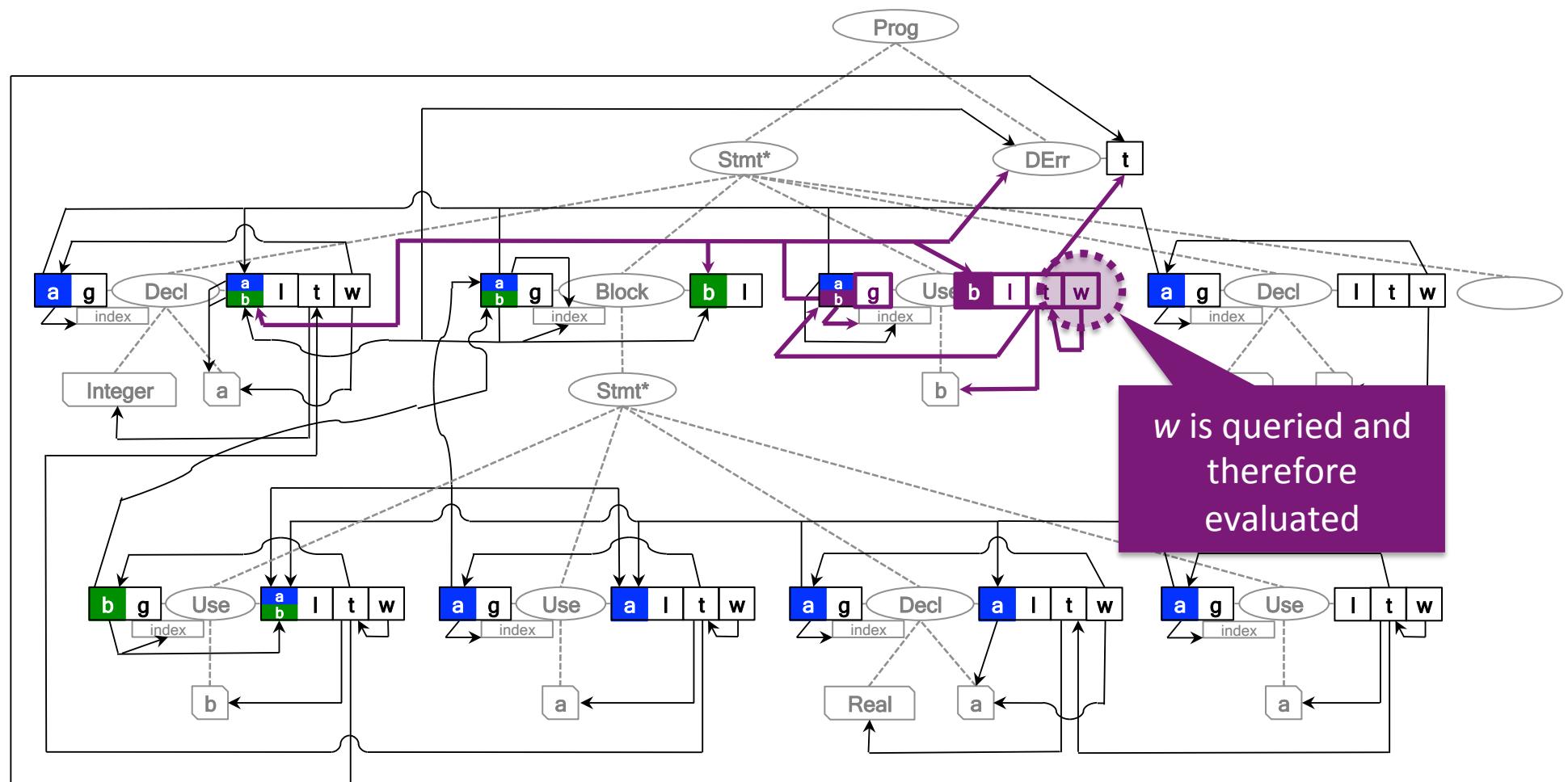
Incremental evaluation



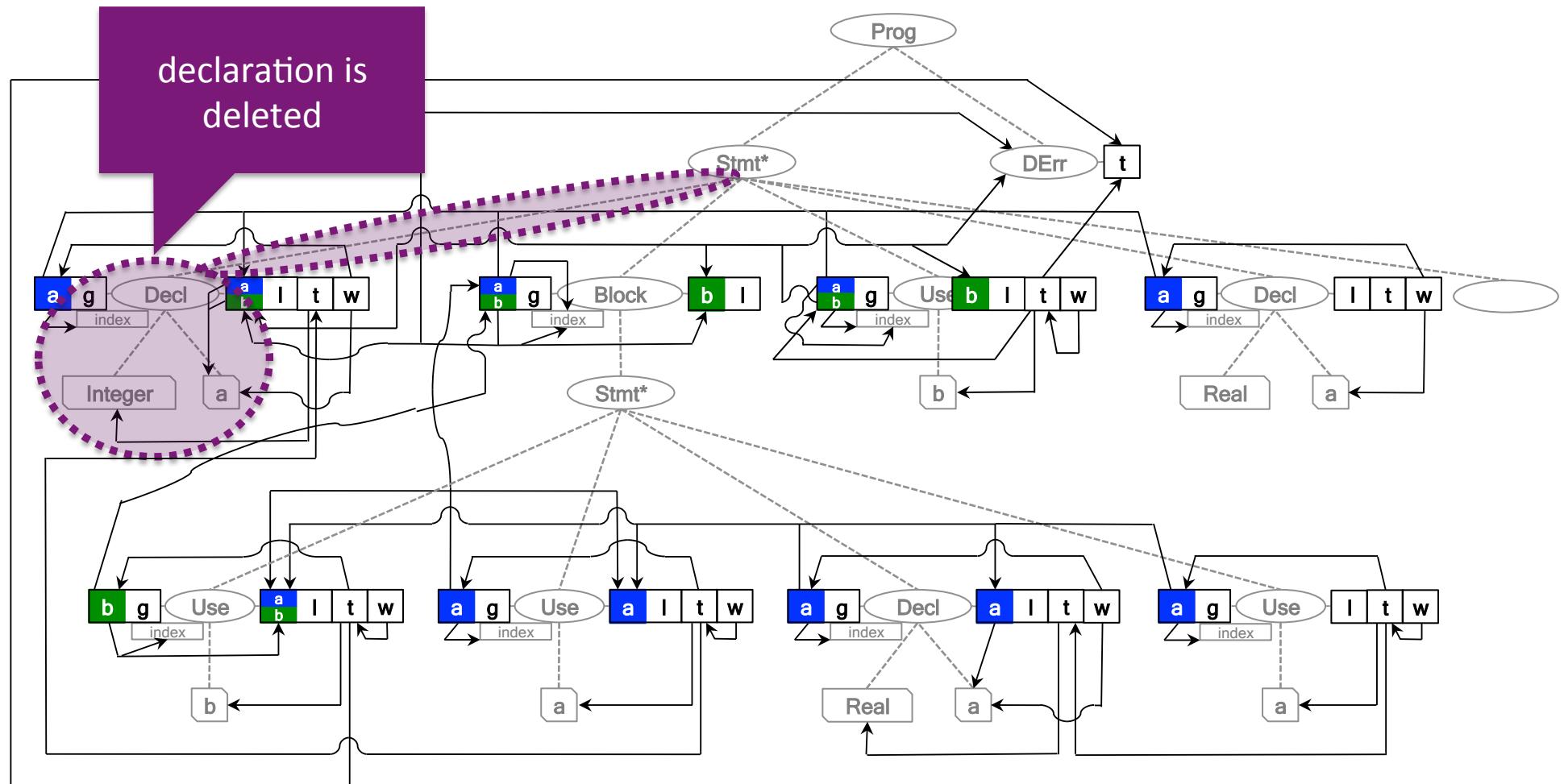
Incremental evaluation



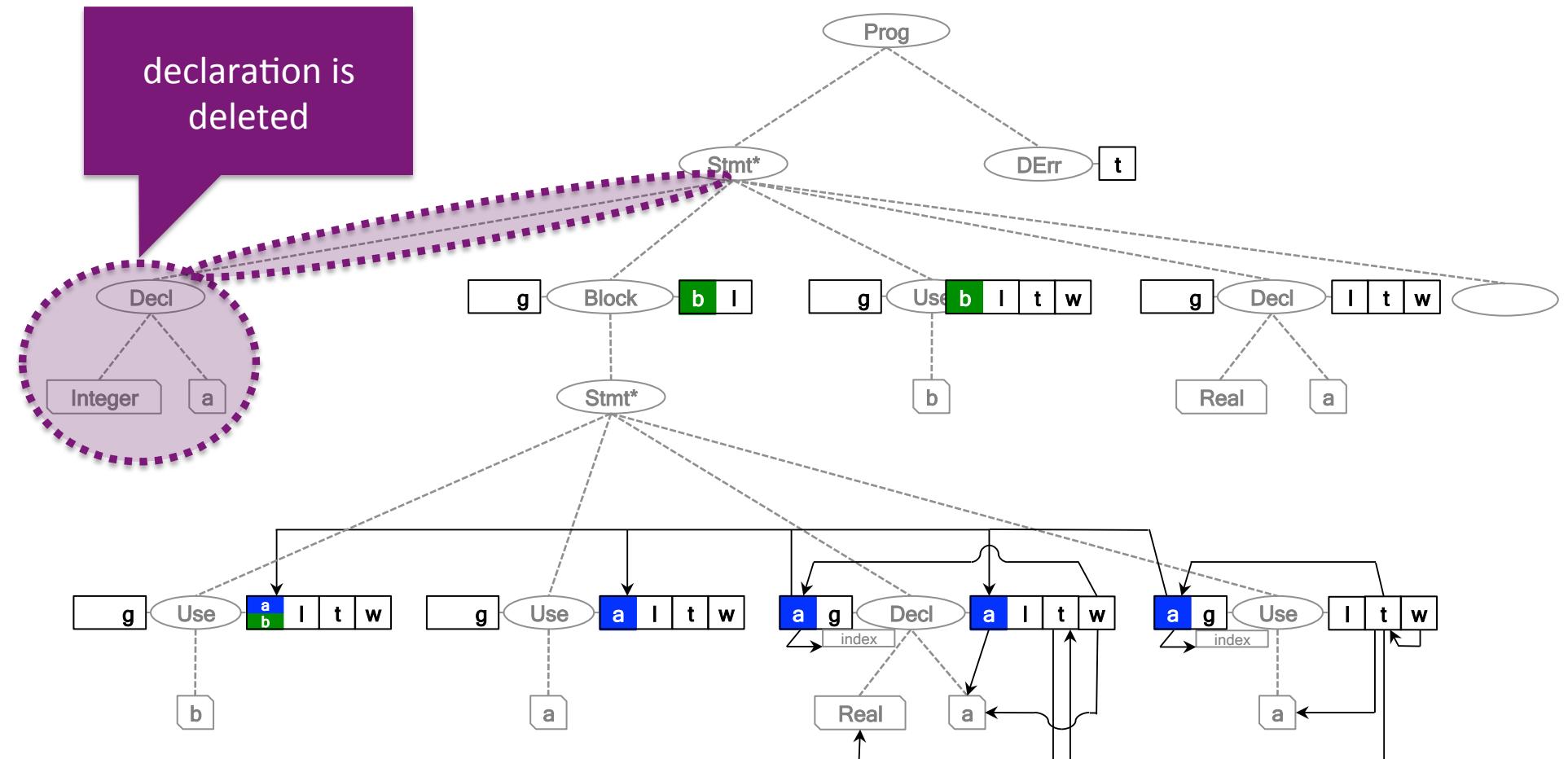
Incremental evaluation



Incremental evaluation



Incremental evaluation



The Application

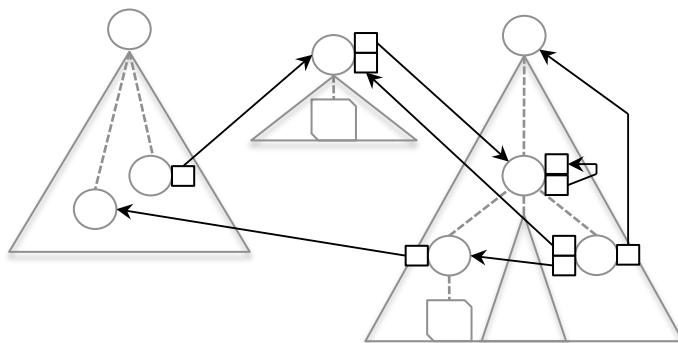
How works RAG-controlled rewriting!

Pattern attributes, transformer
attributes & rewrite deduction

Pattern attributes

Attributes can arbitrary query ASGs:

- including structural relations (reference attributes) and constraints (other attributes)



```
(ag-rule my-pattern ; Pattern attribute  
  (node-type-to-check-pattern-for  
    (lambda (n)  
      ; Query ASG and check constraints.  
      ; Return nodes relevant for rewriting.  
    )))
```

incremental evaluation >>
>> incremental pattern matching

pattern can be deduced
using analyses

Transformer attributes

Attribute values can be functions encapsulating deduced transformations.

```
(ag-rule my-transformation ; Transformer attribute
  (node-type-to-derive-transformation-for
    (lambda (n)
      ; Match fragments to transform (e.g., using pattern attributes).
      (and
        match? ; If transformation is not applicable return false, ...
        (lambda () ; ... otherwise a deduced function encapsulating rewrites.
          ; Apply rewrites on matched fragments.
        )))))
)))))
```

incremental

From programmed through RAG-controlled to ‘wild’ graph rewriting

programmed rewriting via primitive API

```
; Program with arbitrary interleaving of ASG queries & rewrites:  
(let ((c (->child n)  
      (n (if (=conditional-attribute c)  
            (=reference-attribute-1 c) (=reference-attribute-2 c)))  
      (r-subtree n some-new-fragment)))
```

RAG-controlled rewriting

```
; Interactive use of pattern & transformer attributes:  
(let ((trans? (find (lambda (n) (=transformer n)) nodes))  
      (and trans? (trans?))))
```

wild rewriting (fixpoint)

```
; Use generic graph rewriter with transformer attributes:  
(rewrite-all ‘bottom-up list-of-transformer-attributes ASG)
```

all forms supported by *RACR*

The Evaluation

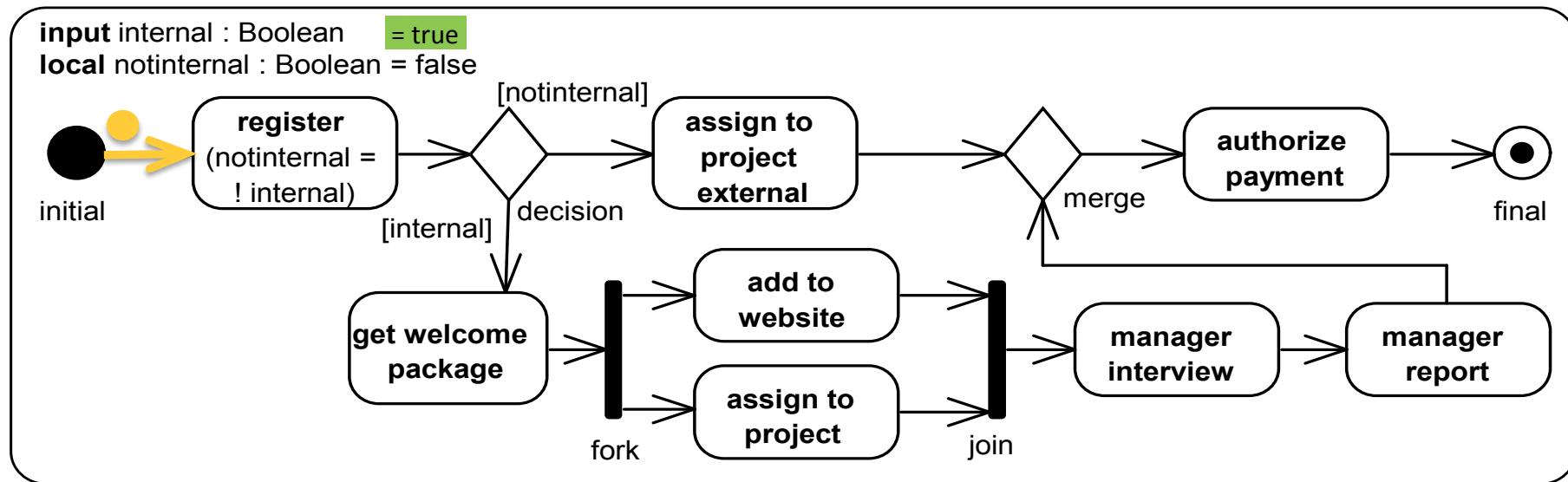
What is your proof of concept?

*fUML Activity Diagrams¹ of TTC 2015,
questionnaires¹ of LWC 2013,
energy auto-tuning case study*

¹ <https://github.com/christoff-buerger/racr>

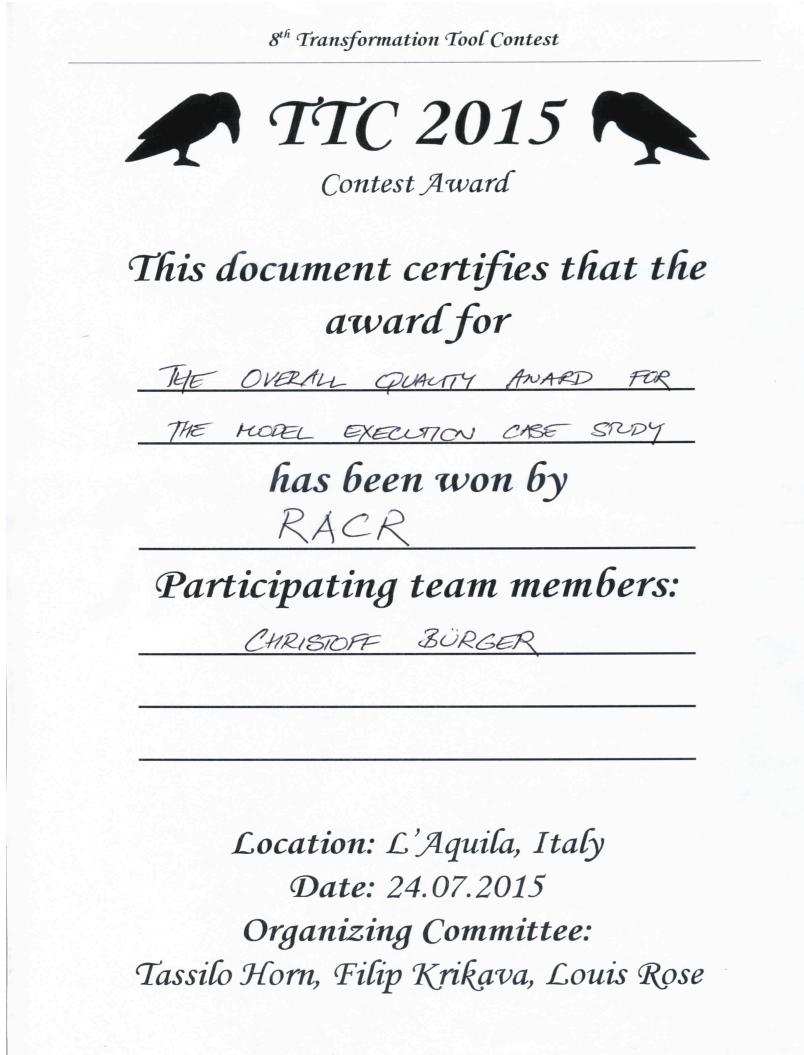
8th Transformation Tool Contest

Task: execution of *fUML Activity Diagrams*.



RACR solution: use enabled analyses to guide incremental state transformations.

8th Transformation Tool Contest



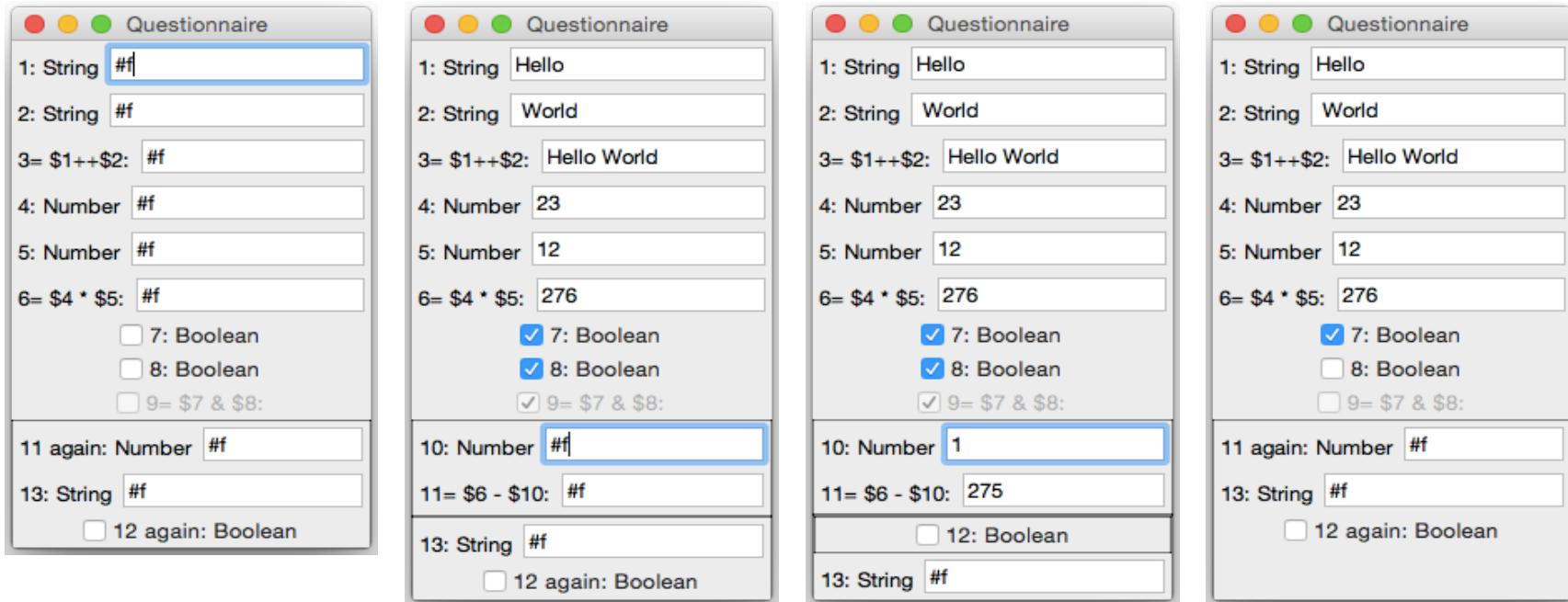
Poster at the poster session

Reference

Christoff Bürger

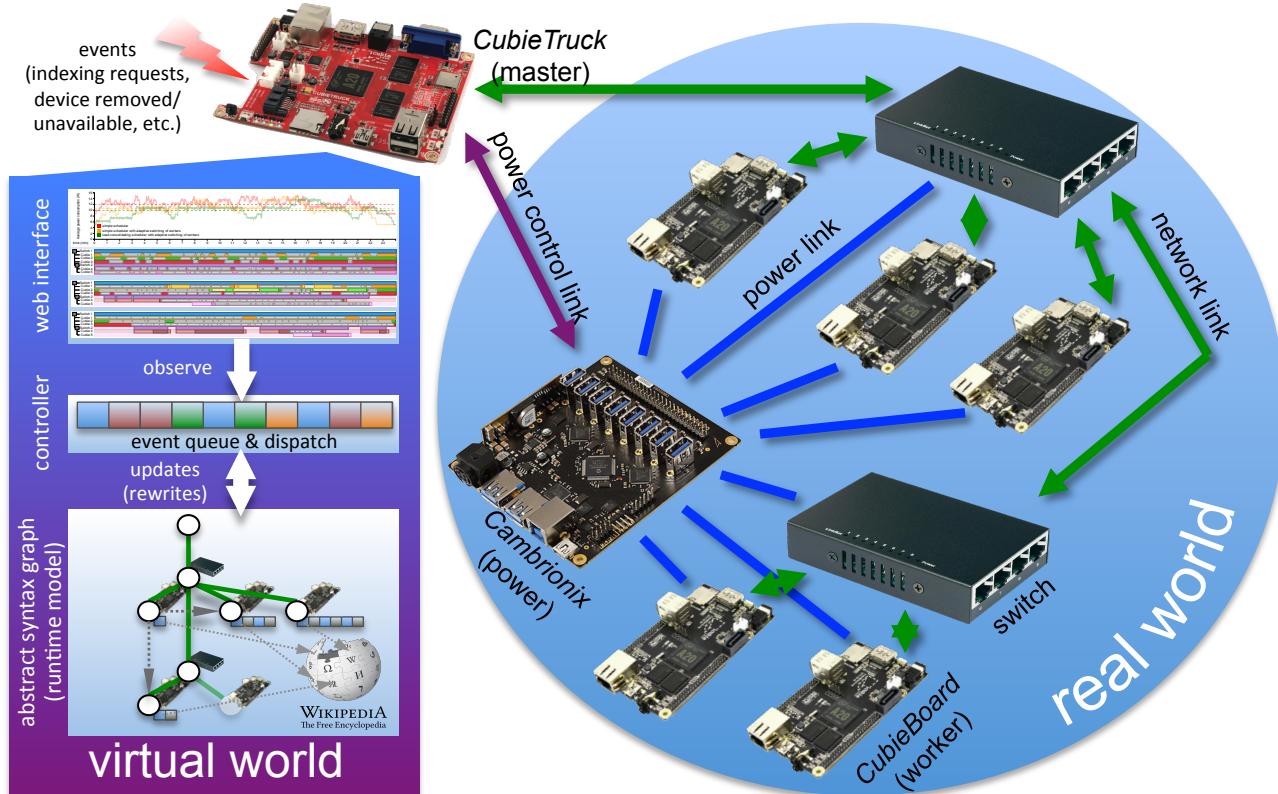
*fUML ACTIVITY DIAGRAMS WITH
RAG-CONTROLLED REWRITING:
A RACR SOLUTION OF THE TTC
2015 MODEL EXECUTION CASE*
CEUR-WS.org, 2015

Language Workbench Challenge 2013



RACR solution: incremental update of computed values & rendering.

Energy auto-tuning case study



Reference

Christoff Bürger et al.
USING REFERENCE ATTRIBUTE GRAMMAR-CONTROLLED REWRITING FOR ENERGY AUTO-TUNING

10th International
Workshop on
Models@run.time,
CEUR-WS.org, 2015

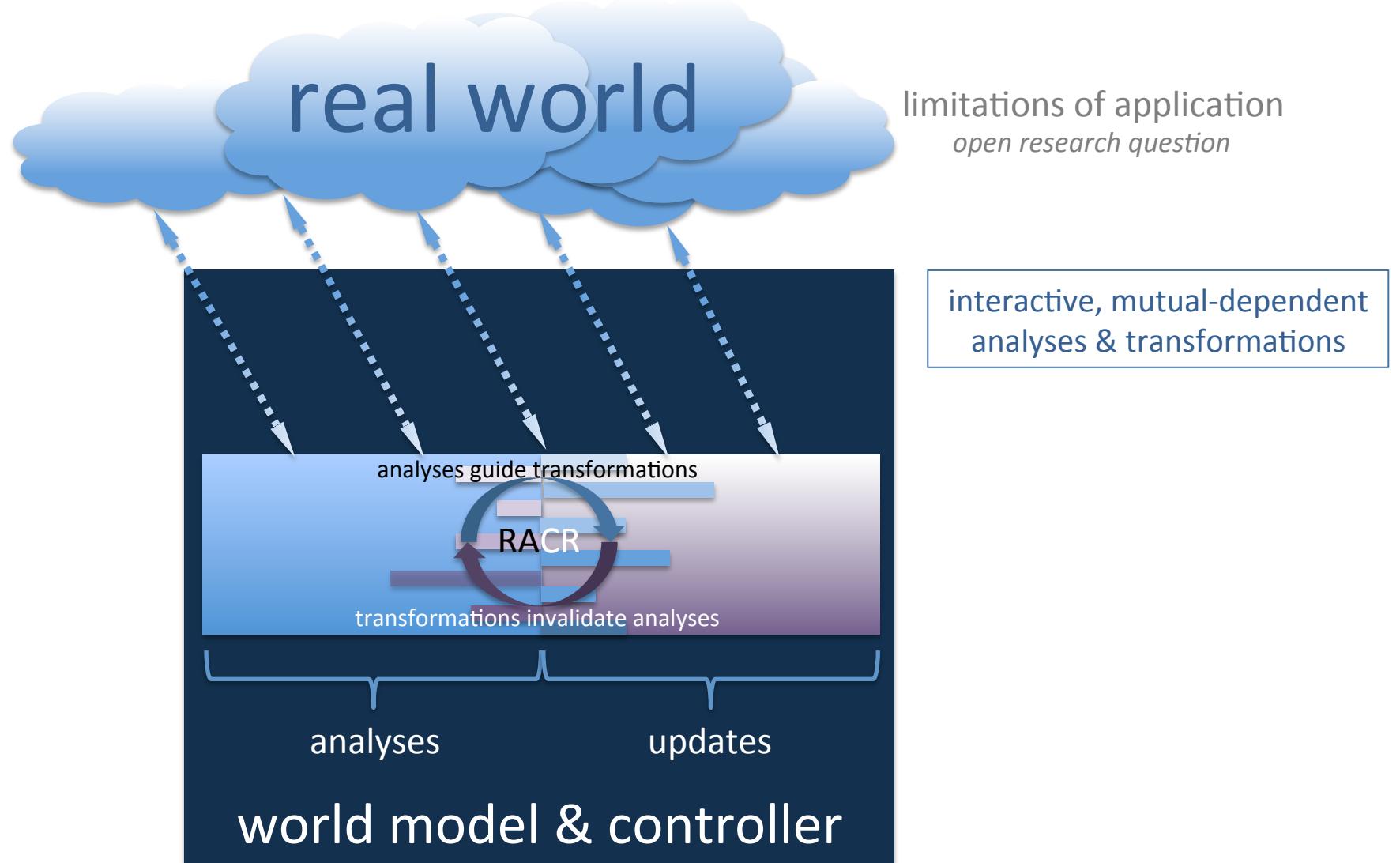
RACR solution: incremental energy efficient scheduling of indexing tasks.

The Intention

What are you up to?

RAG-controlled rewriting for
incremental runtime models

Intended application: runtime models



The Conclusion

What was it all about?

RAG-controlled rewriting enables incremental,
interactive, mutual-dependent analyses and
transformations

What was it all about?

RAG-controlled rewriting

- enables interactive, mutual-dependent ANALYSES and TRANSFORMATIONS
- by seamlessly combining REFERENCE ATTRIBUTE GRAMMARS and GRAPH REWRITING
 - such that ANALYSES CAN GUIDE AND DEDUCE REWRITES
 - and REWRITES UPDATE ANALYSES they influence
- using a well-balanced set of QUERY- and REWRITE-FUNCTIONS
 - constructing a DYNAMIC ATTRIBUTE DEPENDENCY GRAPH
 - that can be used for DYNAMIC ATTRIBUTE INVALIDATION
 - achieving INCREMENTAL ANALYSES AND TRANSFORMATIONS

incremental, interactive, mutual-dependent analyses and transformations

Backup slides

Dynamic dependency over-
approximations

Dynamic dependency over-approximations

