Reference attribute grammar controlled graph rewriting

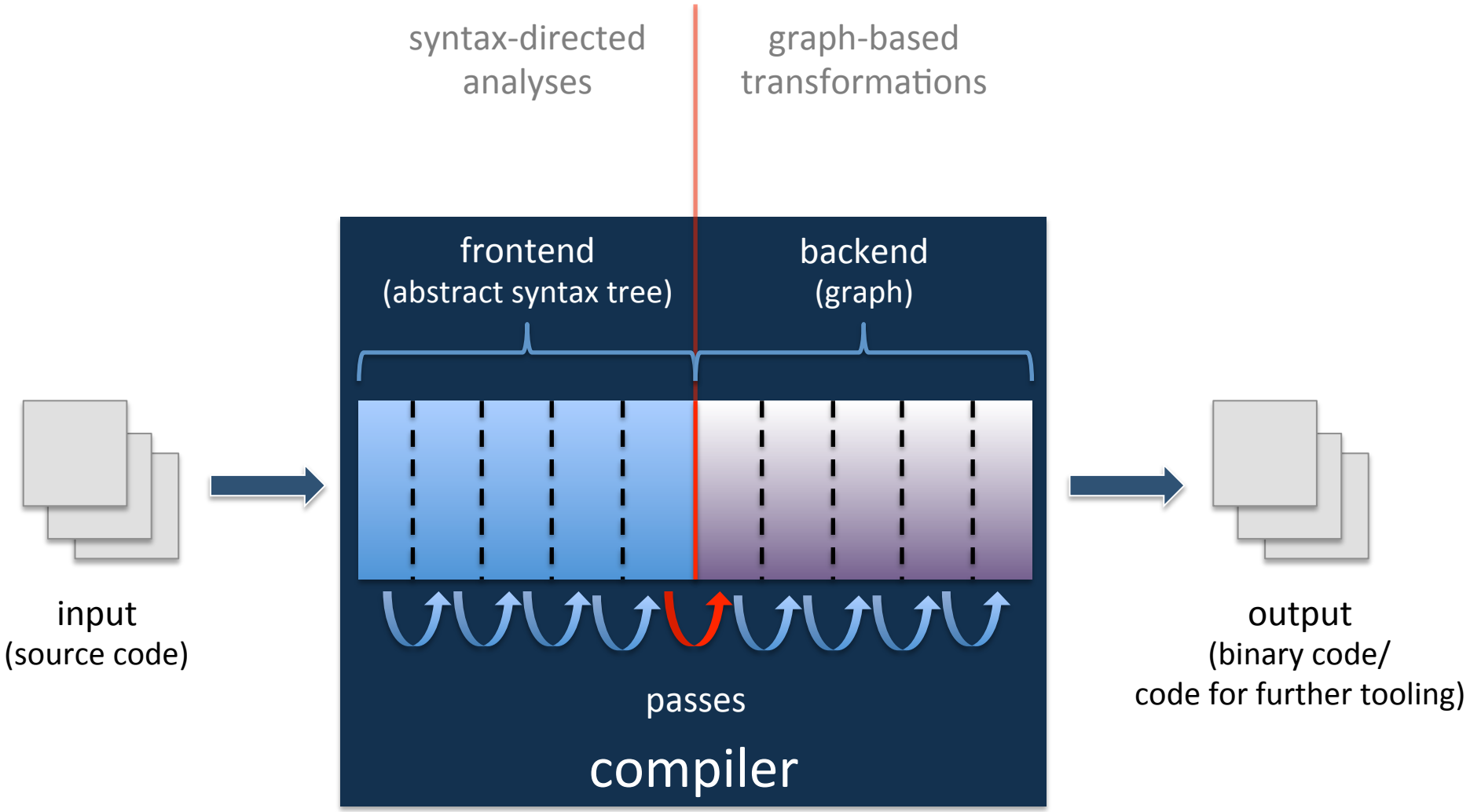
Motivation and overview

Christoff Bürger
christoff.buerger@gmail.com

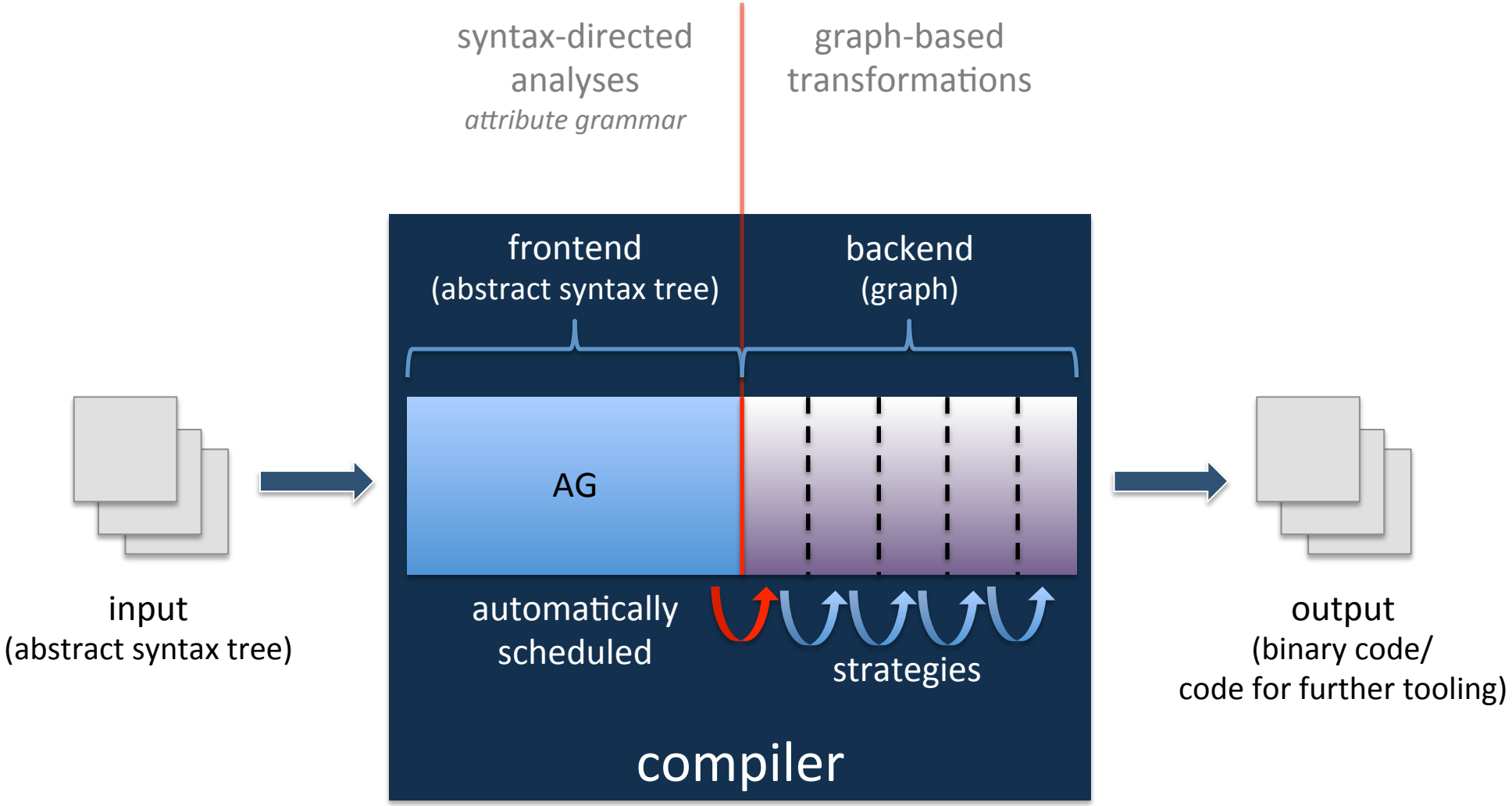
The problem
What do you want?

Interactive, mutual-dependent analyses
& transformations

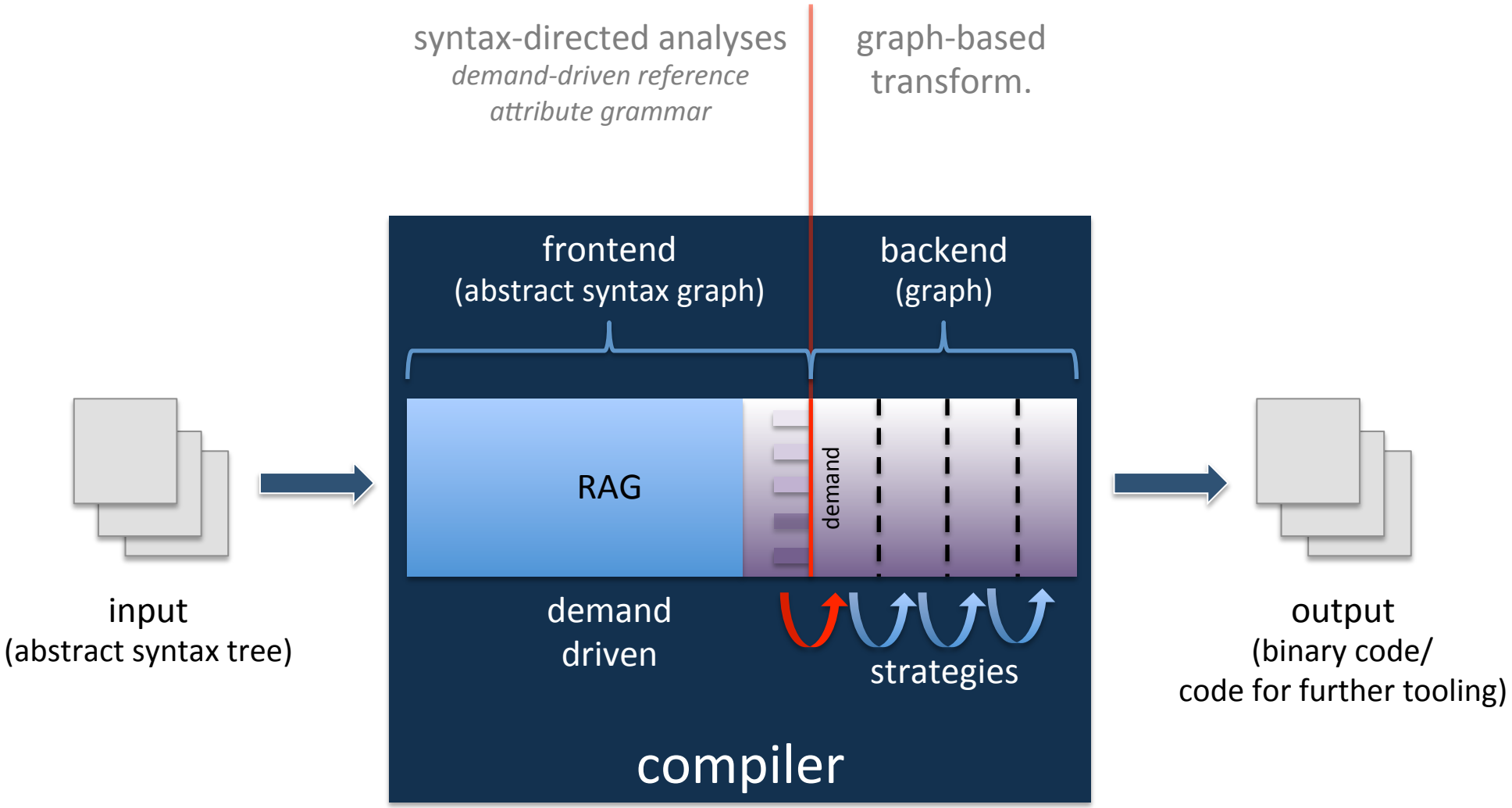
From batch to interactive



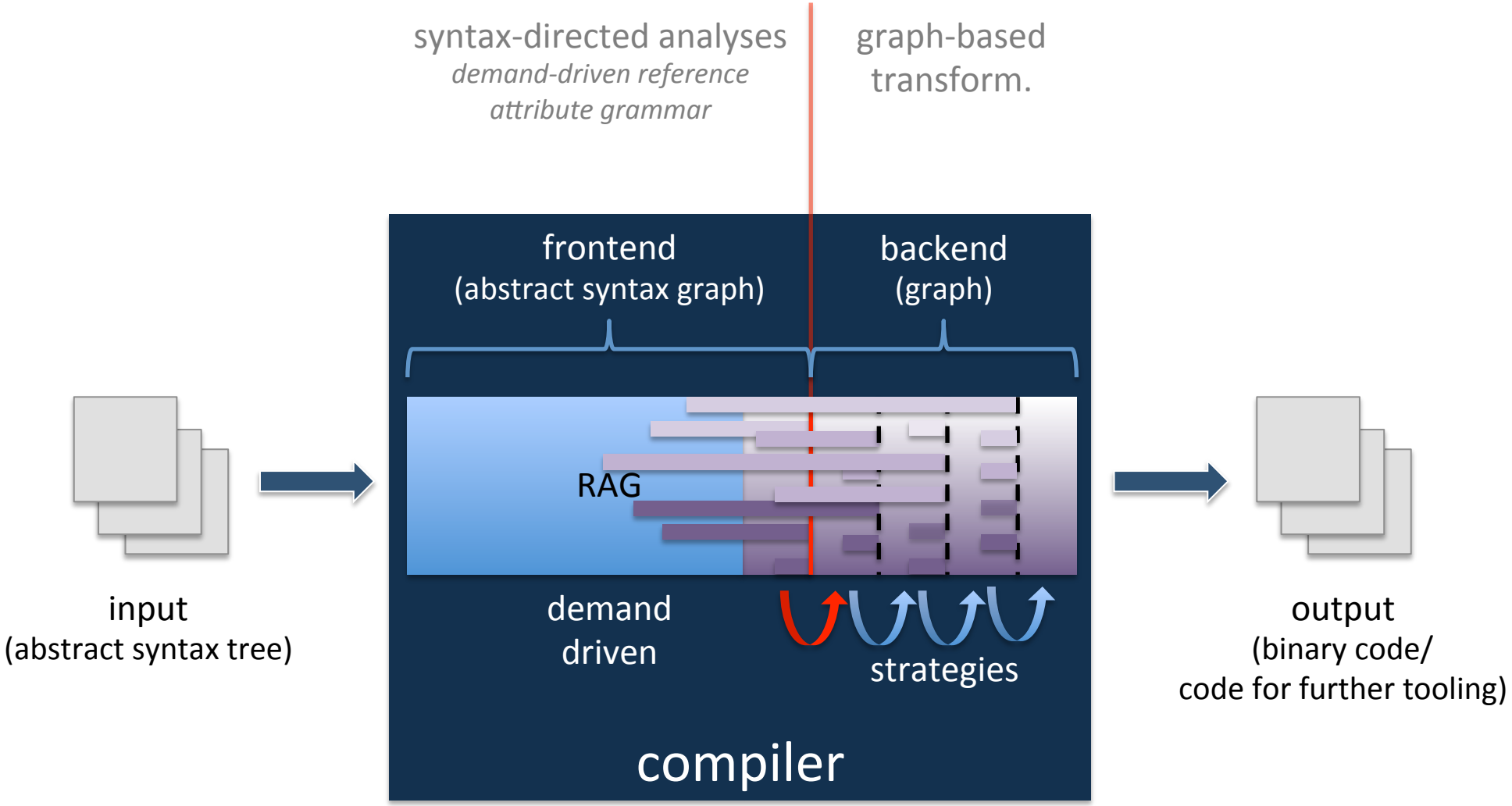
From batch to interactive



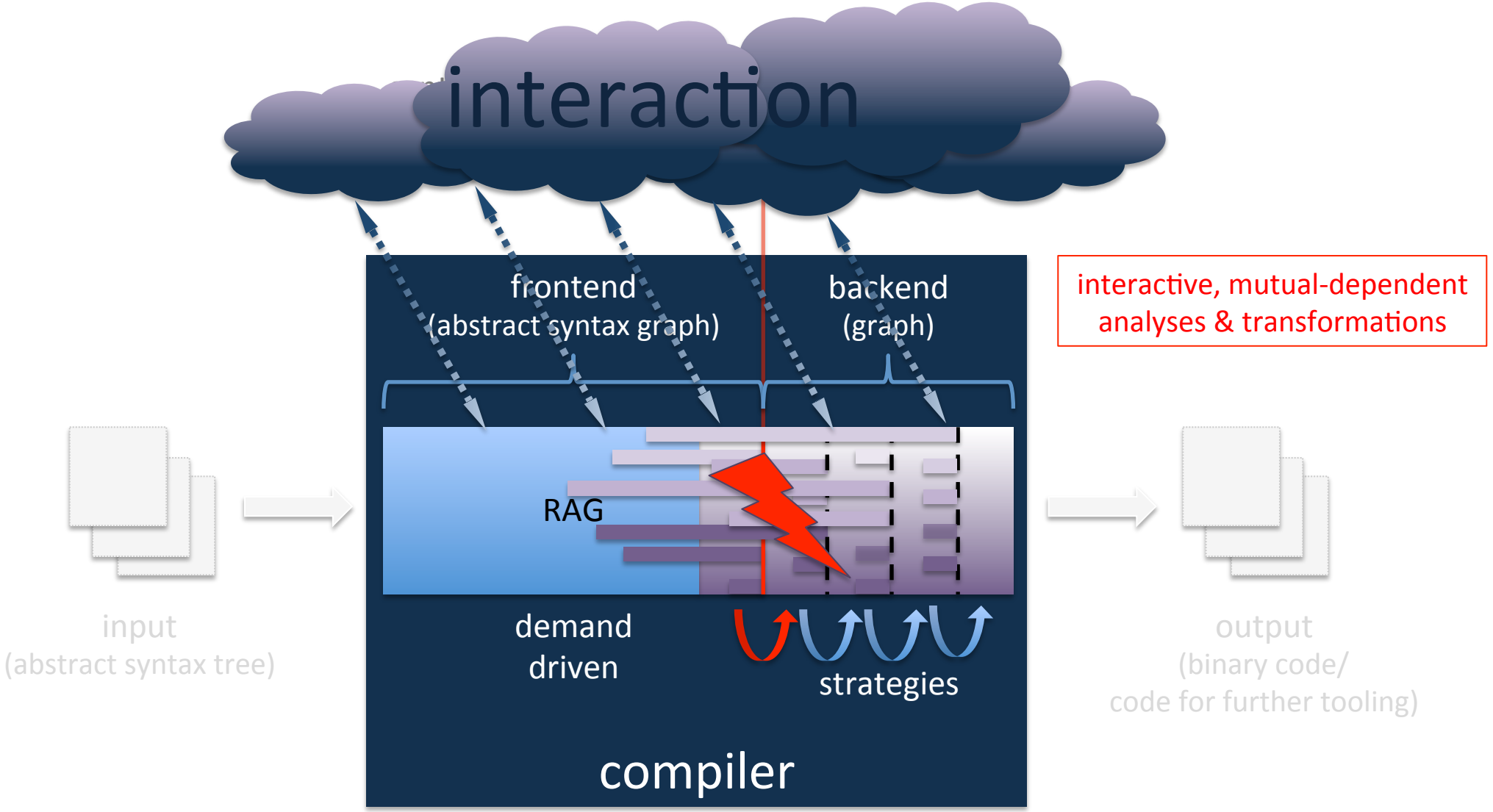
From batch to interactive



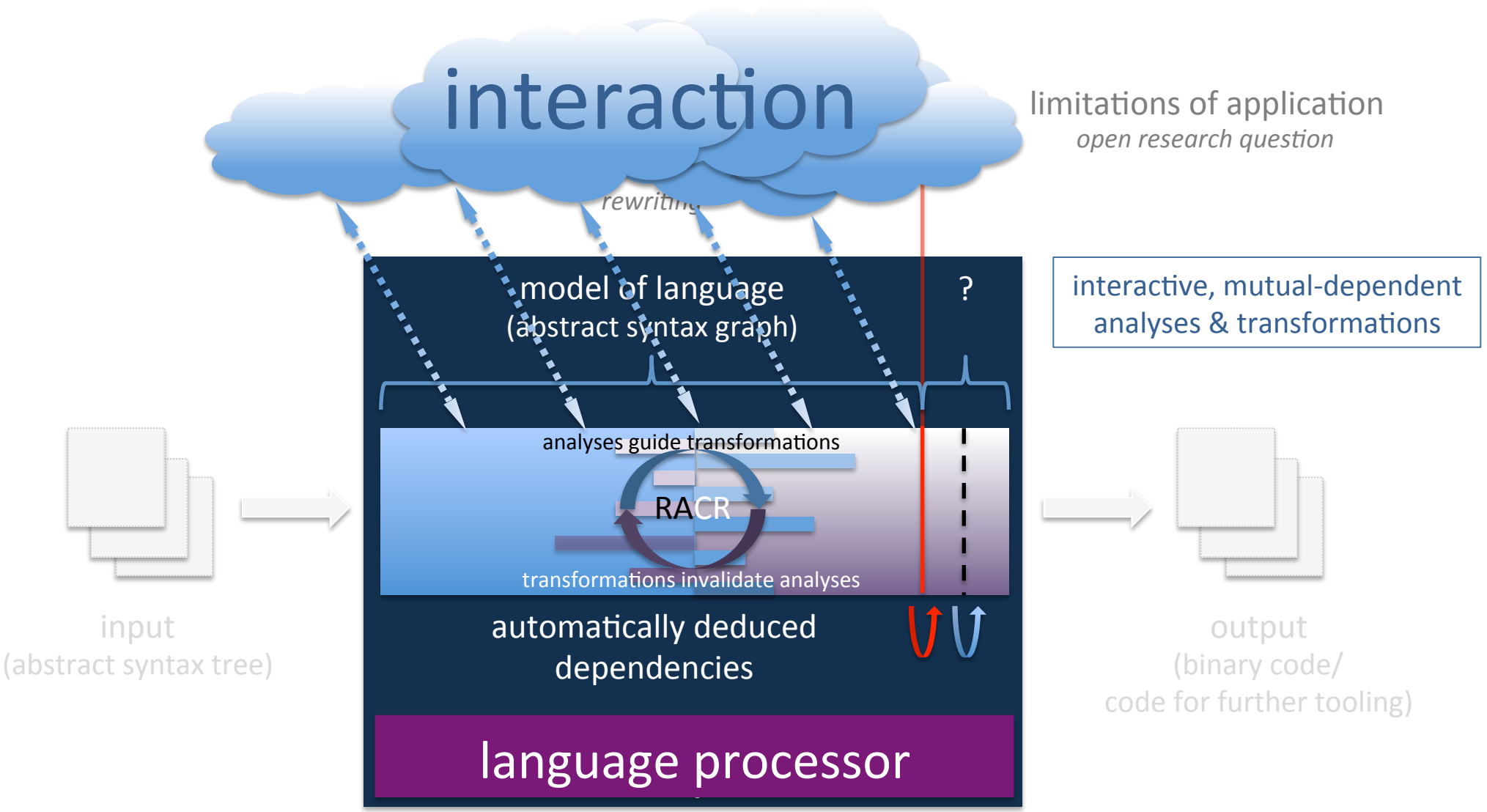
From batch to interactive



From batch to interactive



From batch to interactive



The solution
What is RAG-controlled rewriting?

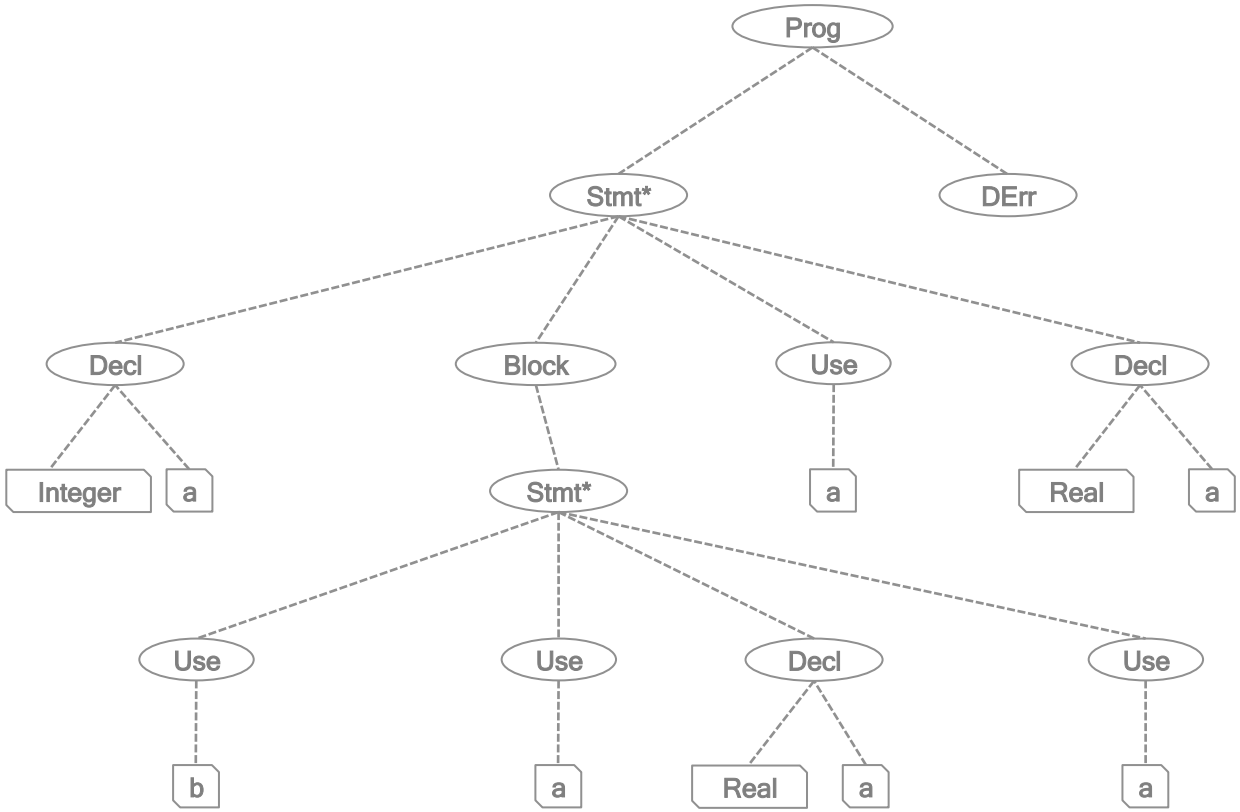
Reference attribute grammars, abstract syntax graphs, RAG-controlled rewriting & *RACR*[1]

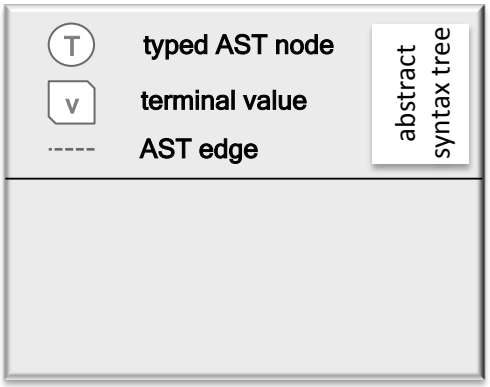
[1] https://github.com/christoff-buerger/racr

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
```

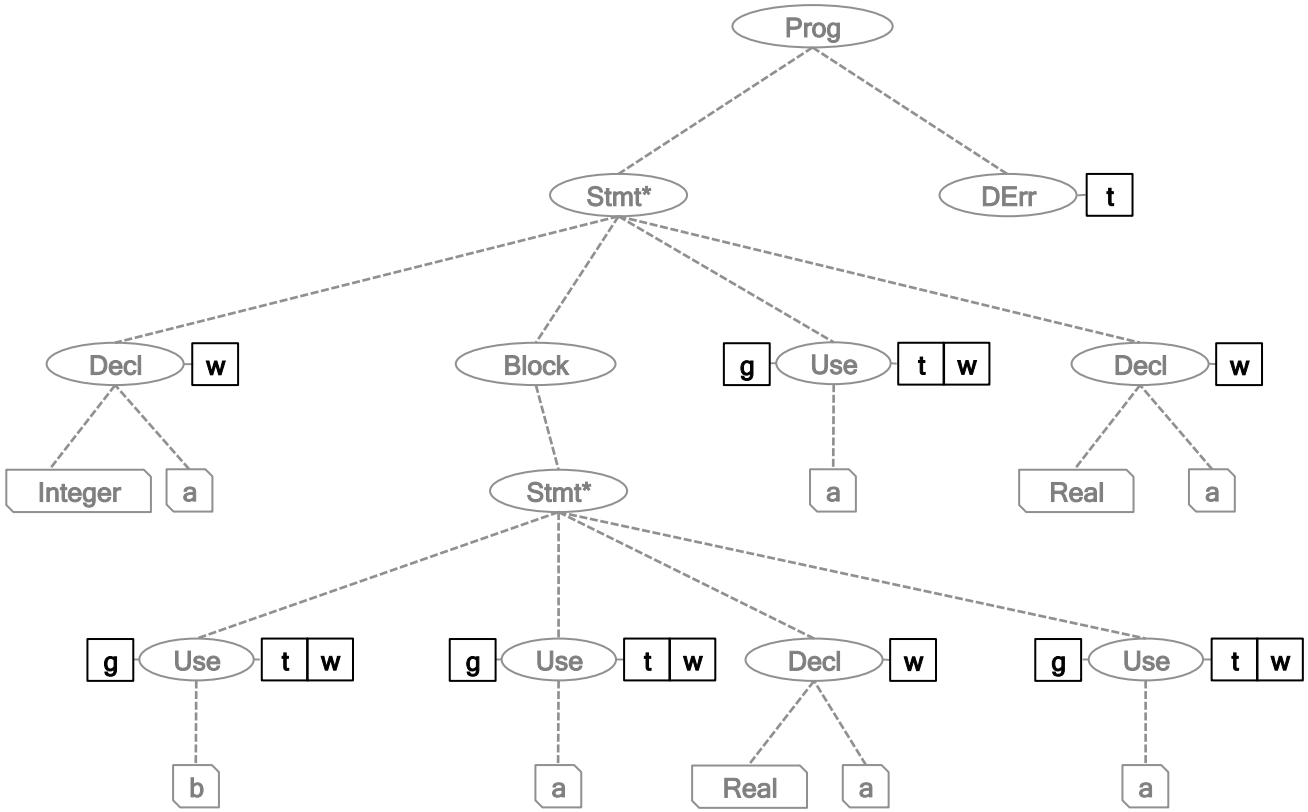
Reference attribute grammars & ASGs

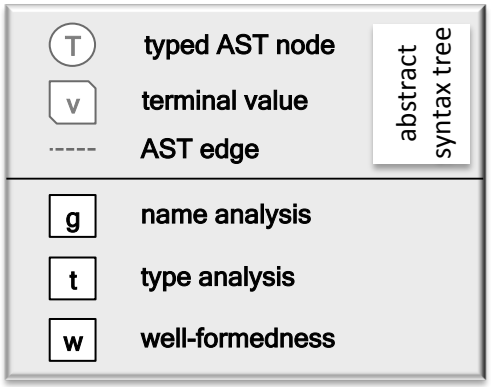





```
Program
        decl a : integer
        Begin
                use b ; Error
                use a
                decl a : real
                use a
        End
        use a
        decl a : real ; Error
End
```

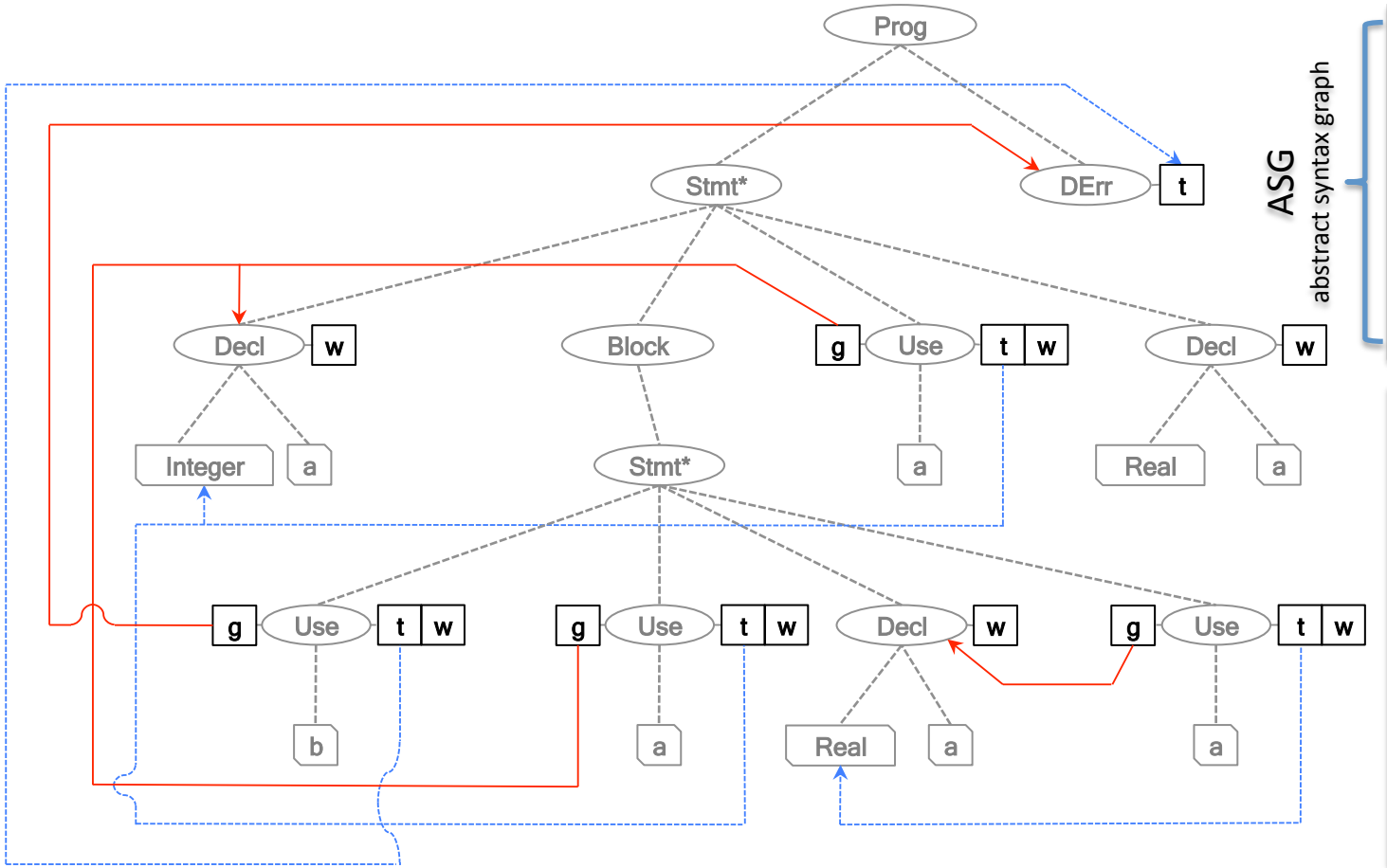
Reference attribute grammars & ASGs

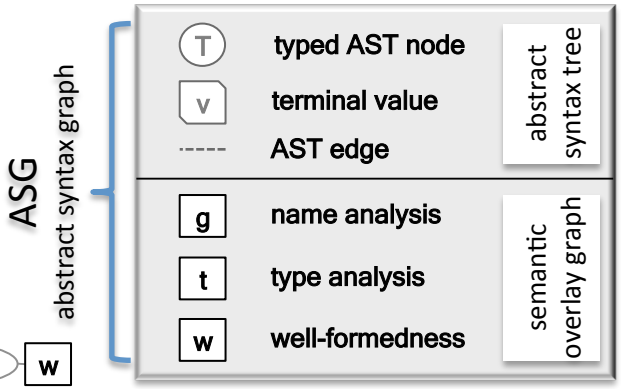





```
Program
        decl a : integer
        Begin
                use b ; Error
                use a
                decl a : real
                use a
        End
        use a
        decl a : real ; Error
End
```

Reference attribute grammars & ASGs






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

RAG-controlled rewriting

- RAG-controlled rewriting = RAGs + rewriting
  - RAG for declarative analyses
  - graph rewriting for declarative 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

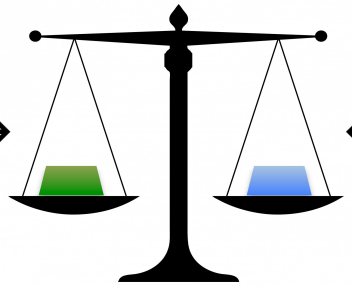
The implementation
How works RAG-controlled rewriting?

Dynamic attribute dependency graphs &
incremental evaluation

Query & rewrite functions

well balanced set of functions

query functions to access ASG
add dependencies on queried information →

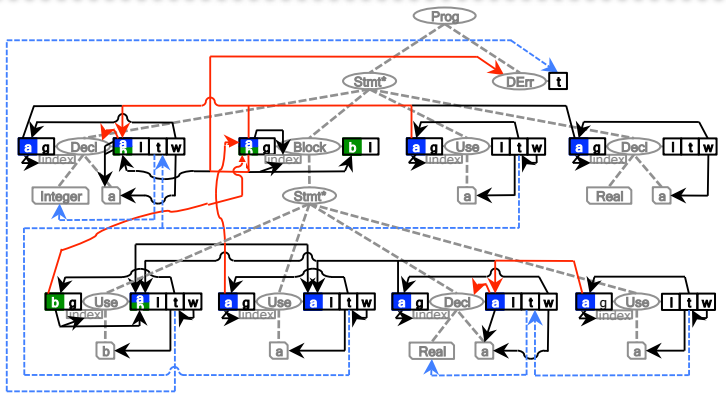


← rewrite functions to change ASG
invalidate attributes depending on rewritten information

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

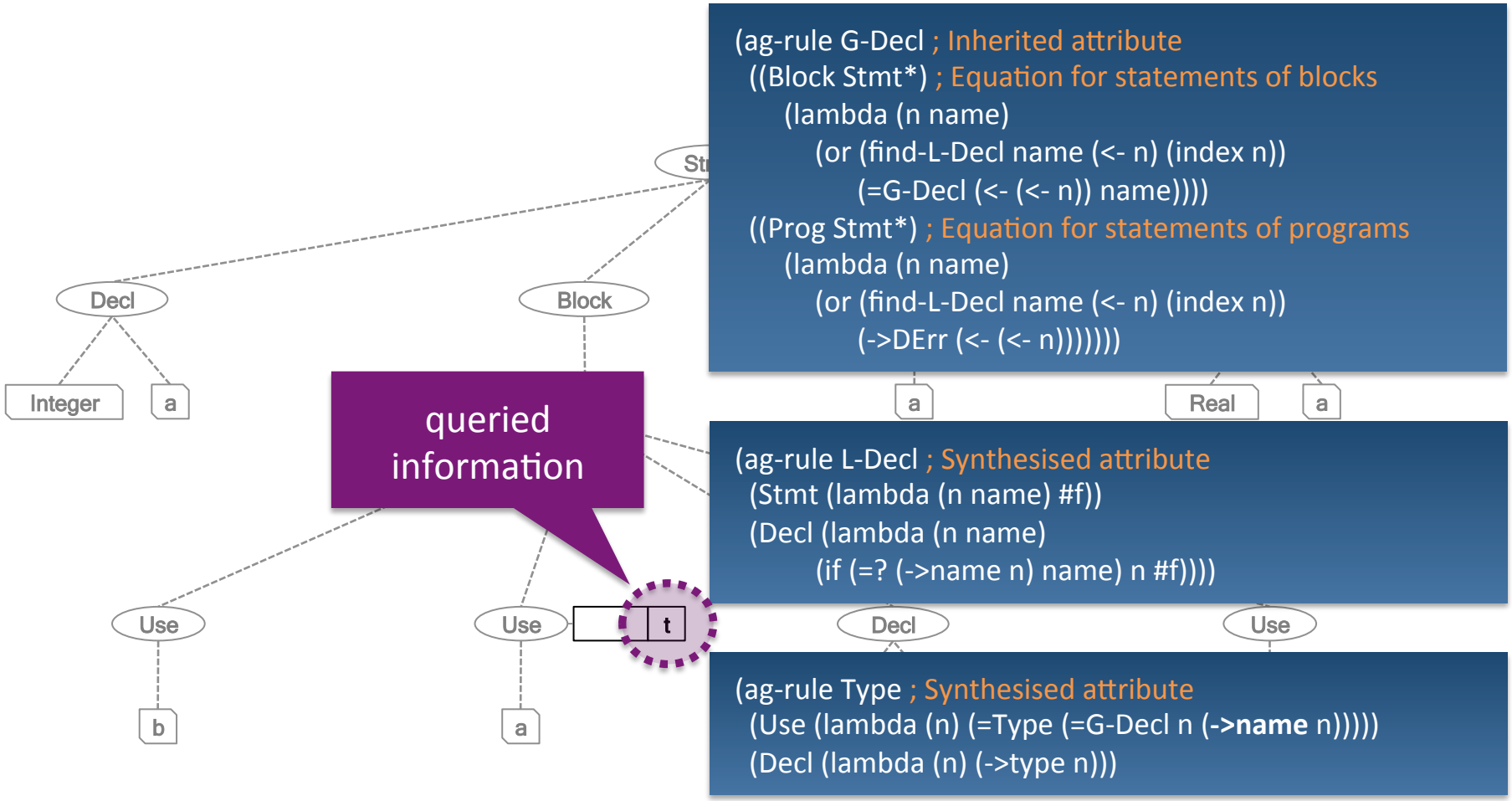
rewrite functions

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

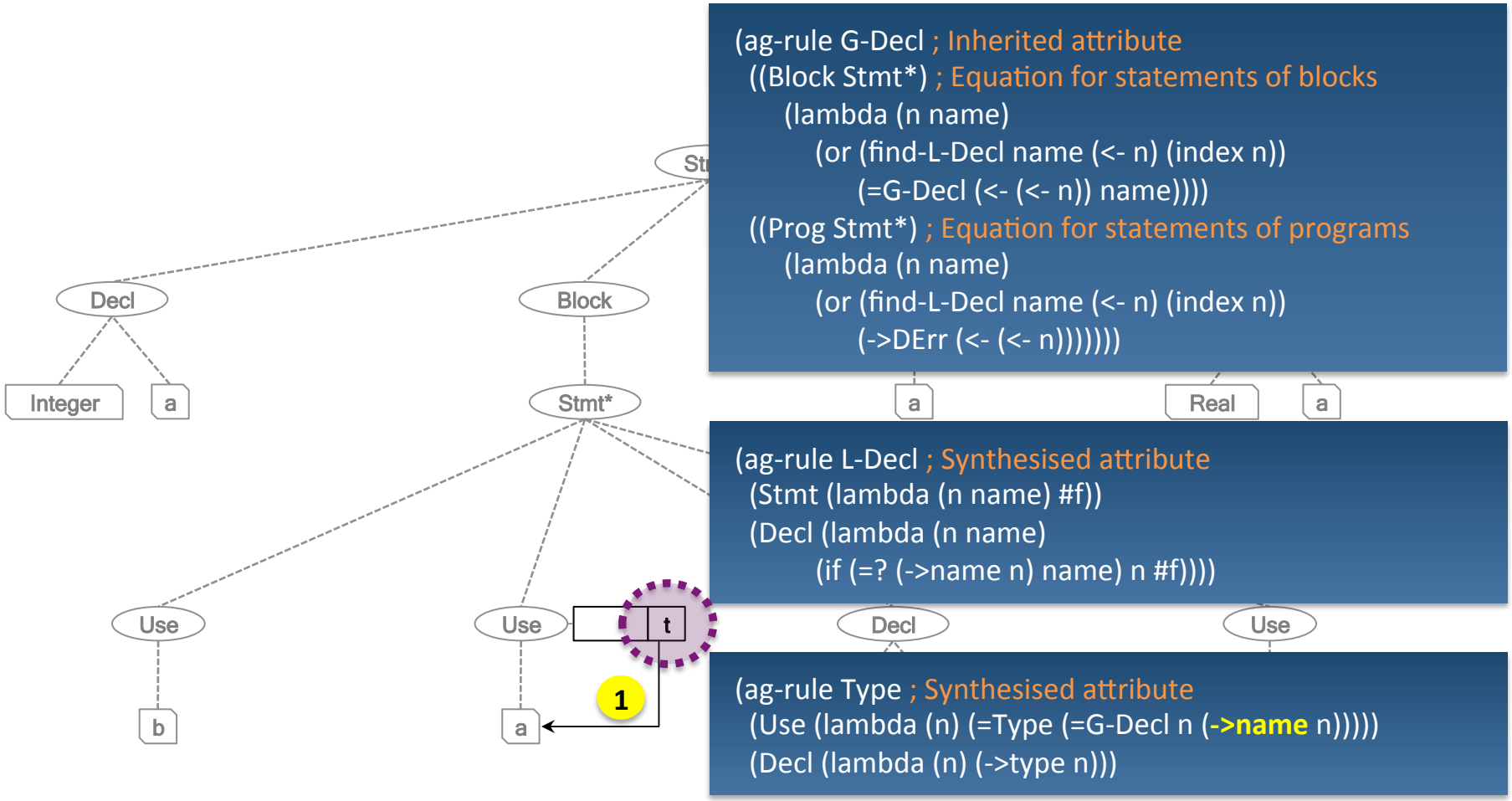
dependency types

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

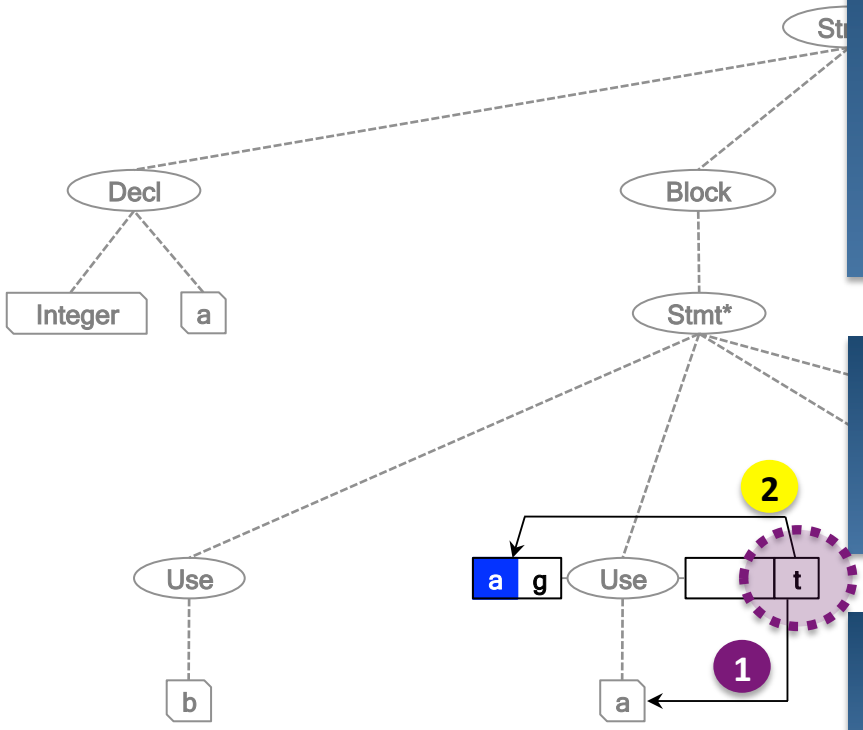
Dynamic attribute dependency graphs



Dynamic attribute dependency graphs



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)))))))
```

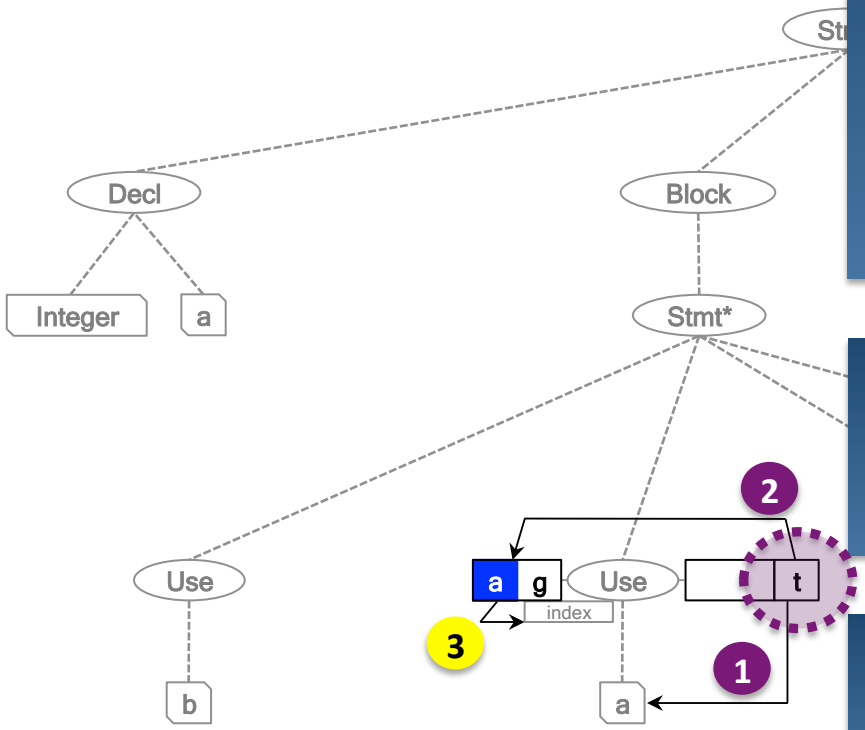
a Real a

```
(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

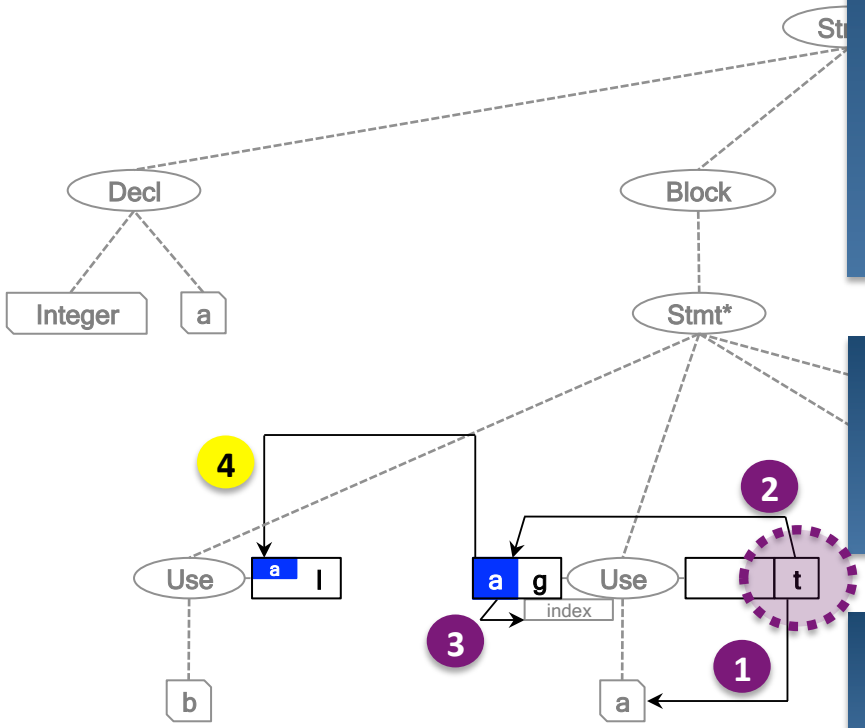



```
(ag-rule G-Decl ; Inherited attribute
 ((Block Stmt*) ; Equation for statements of blocks
    (lambda (n name)
       (or (find-L-Decl name (<- n) (index n))
           (=G-Decl (<- (<- n)) name))))
 ((Prog Stmt*) ; Equation for statements of programs
    (lambda (n name)
       (or (find-L-Decl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

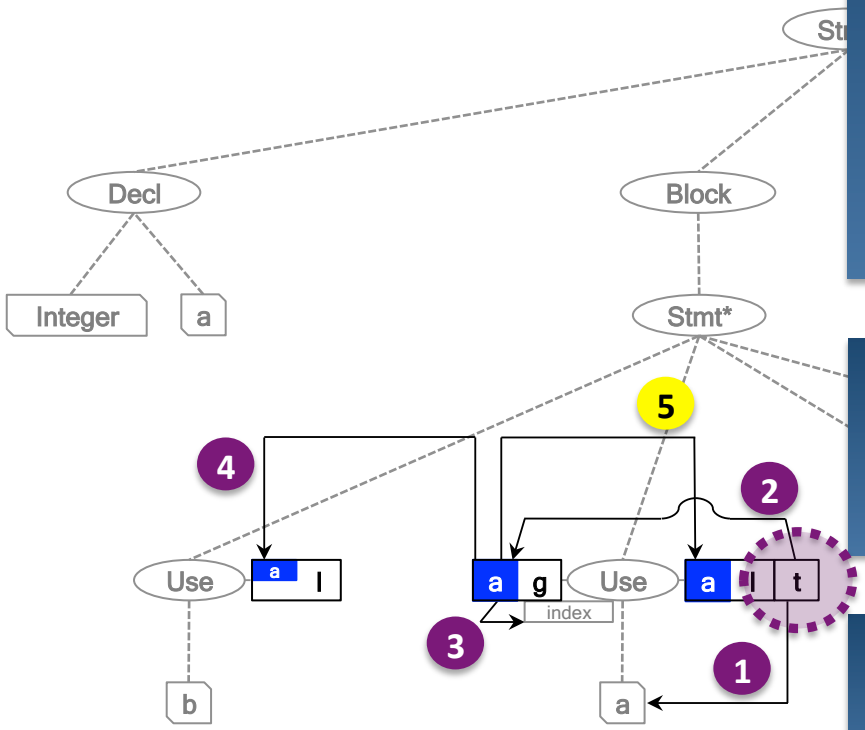
a Real a

```
(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)))))))
```

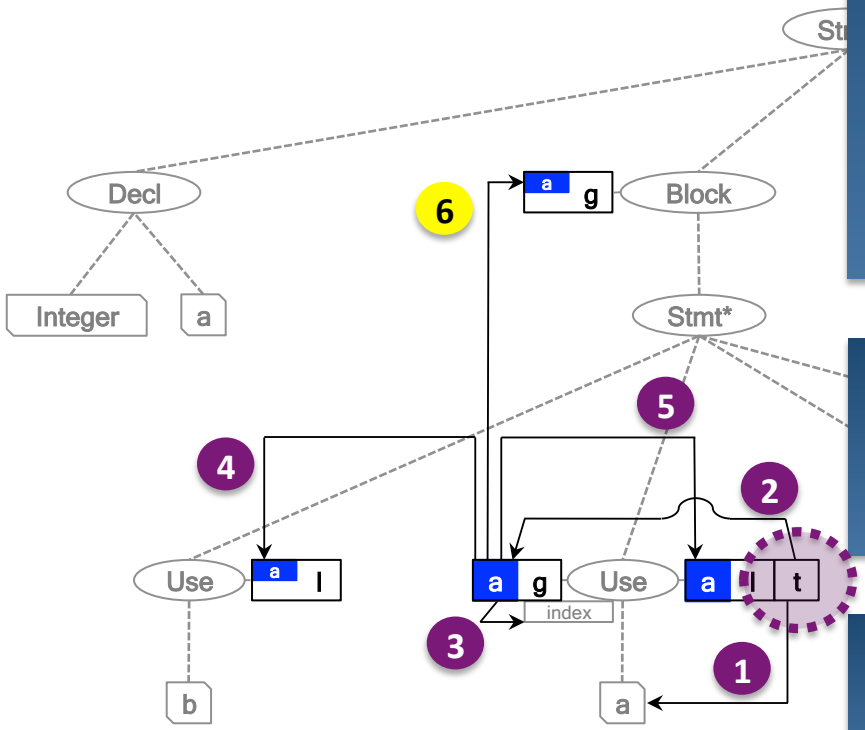
a Real a

```
(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)))))))
```

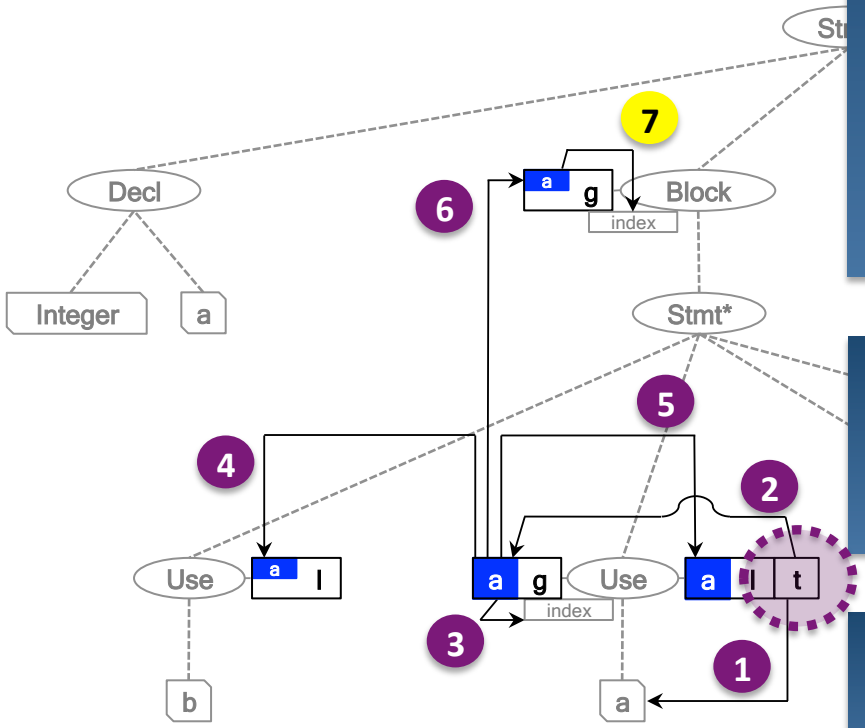
a Real a

```
(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

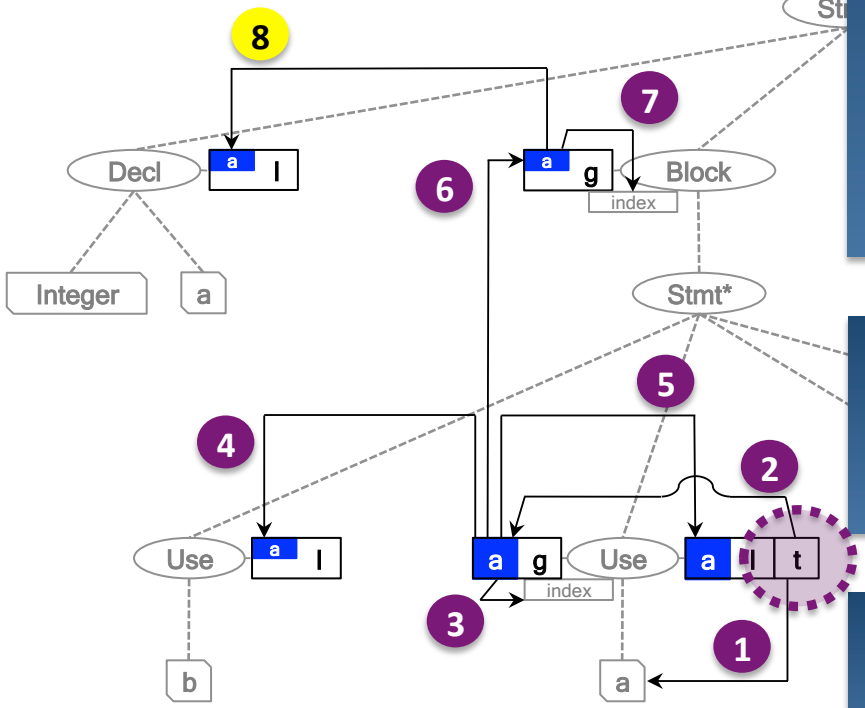



```
(ag-rule G-Decl ; Inherited attribute
  ((Block Stmt*) ; Equation for statements of blocks
     (lambda (n name)
        (or (find-L-Decl name (<- n) (index n))
            (=G-Decl (<- (<- n)) name))))
  ((Prog Stmt*) ; Equation for statements of programs
     (lambda (n name)
        (or (find-L-Decl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

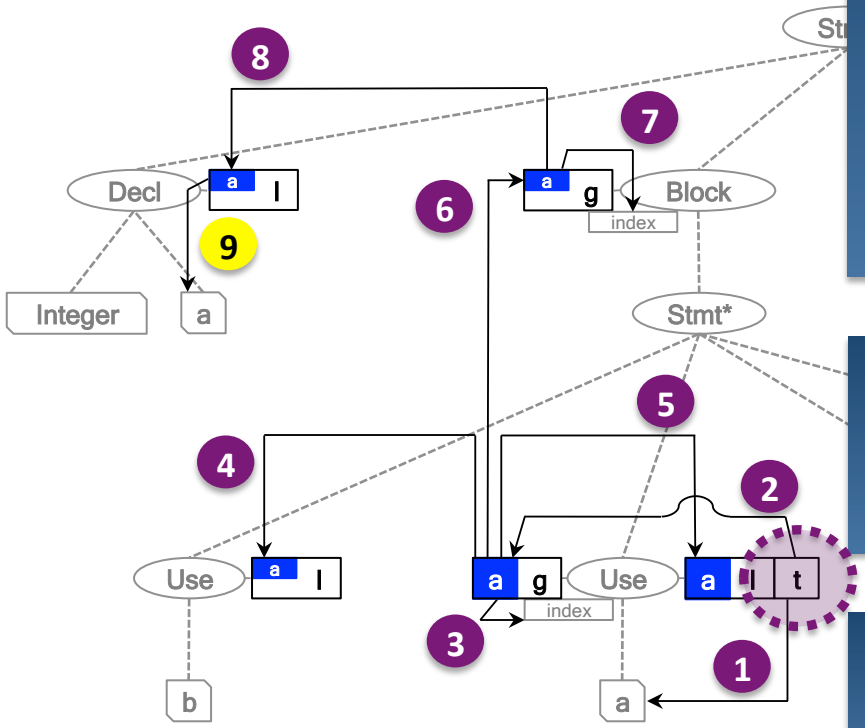
a Real a

```
(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)))))))
```

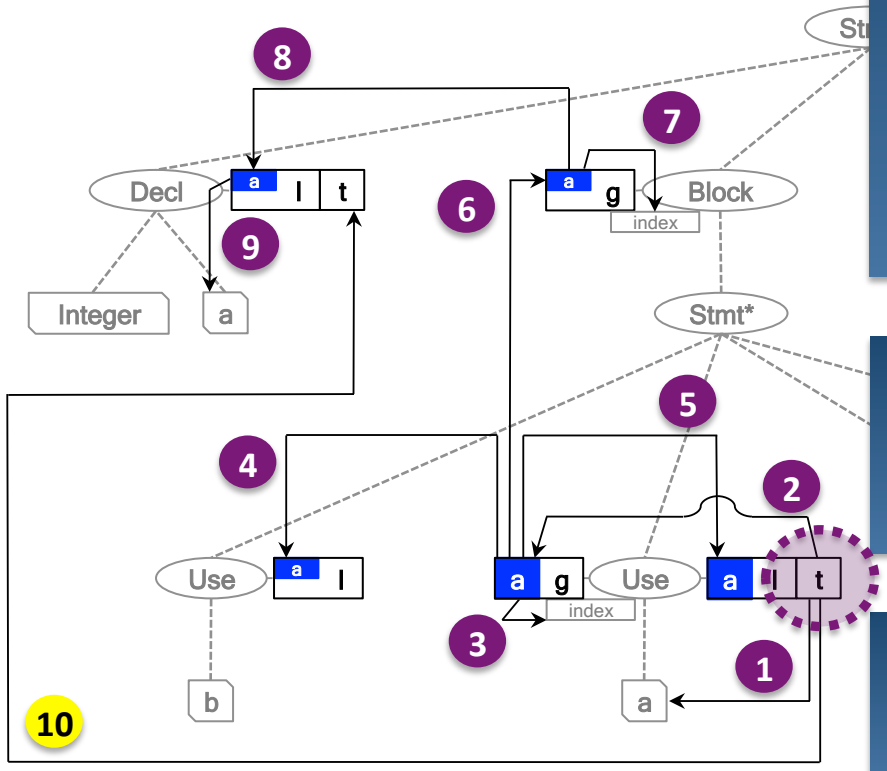
a Real a

```
(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)))))))
```

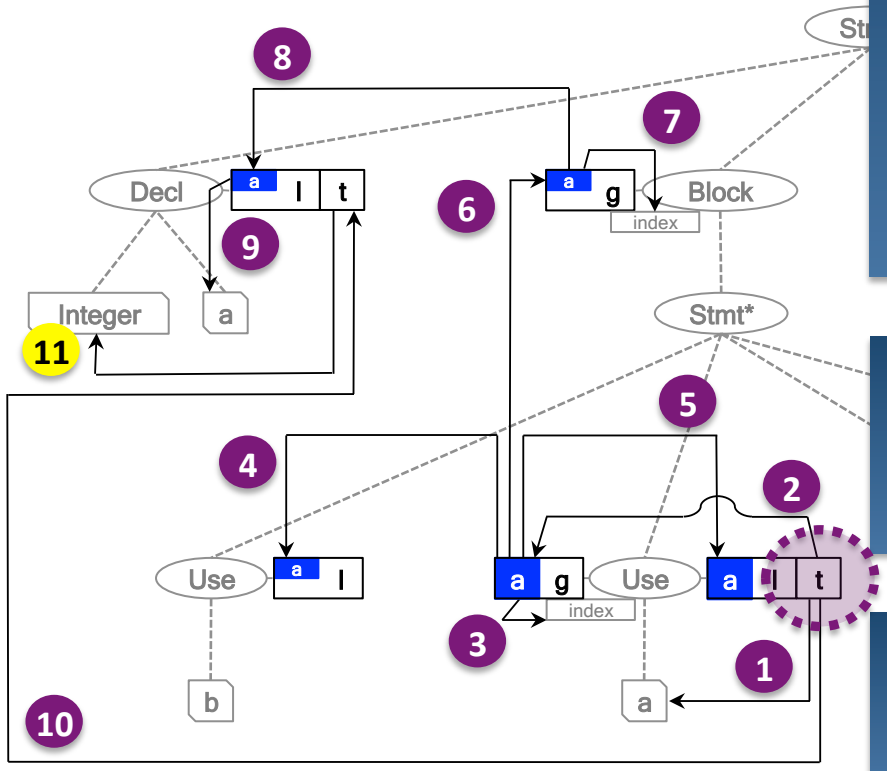
a Real a

```
(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)))))))
```

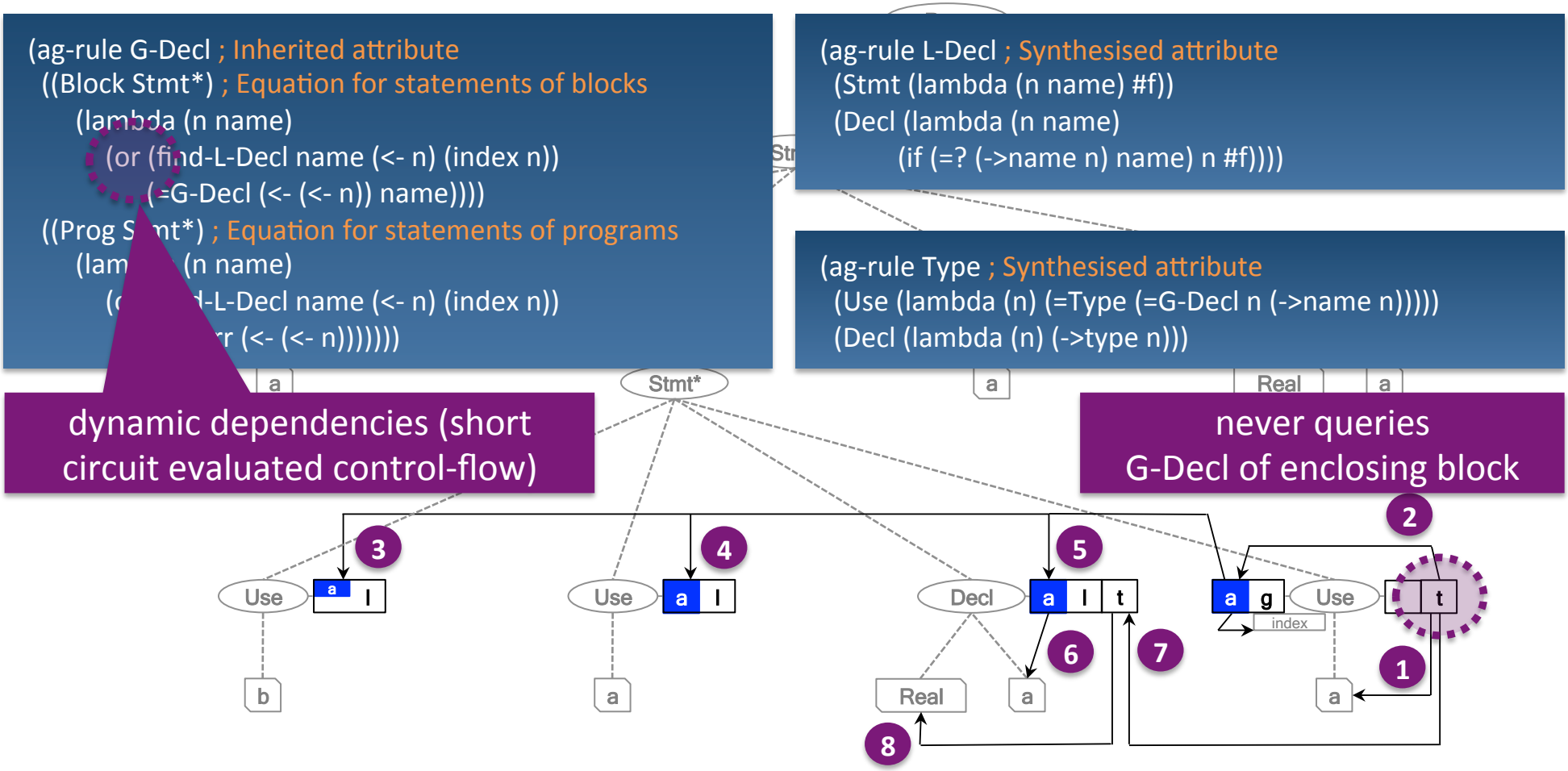
a Real a

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

Decl Use

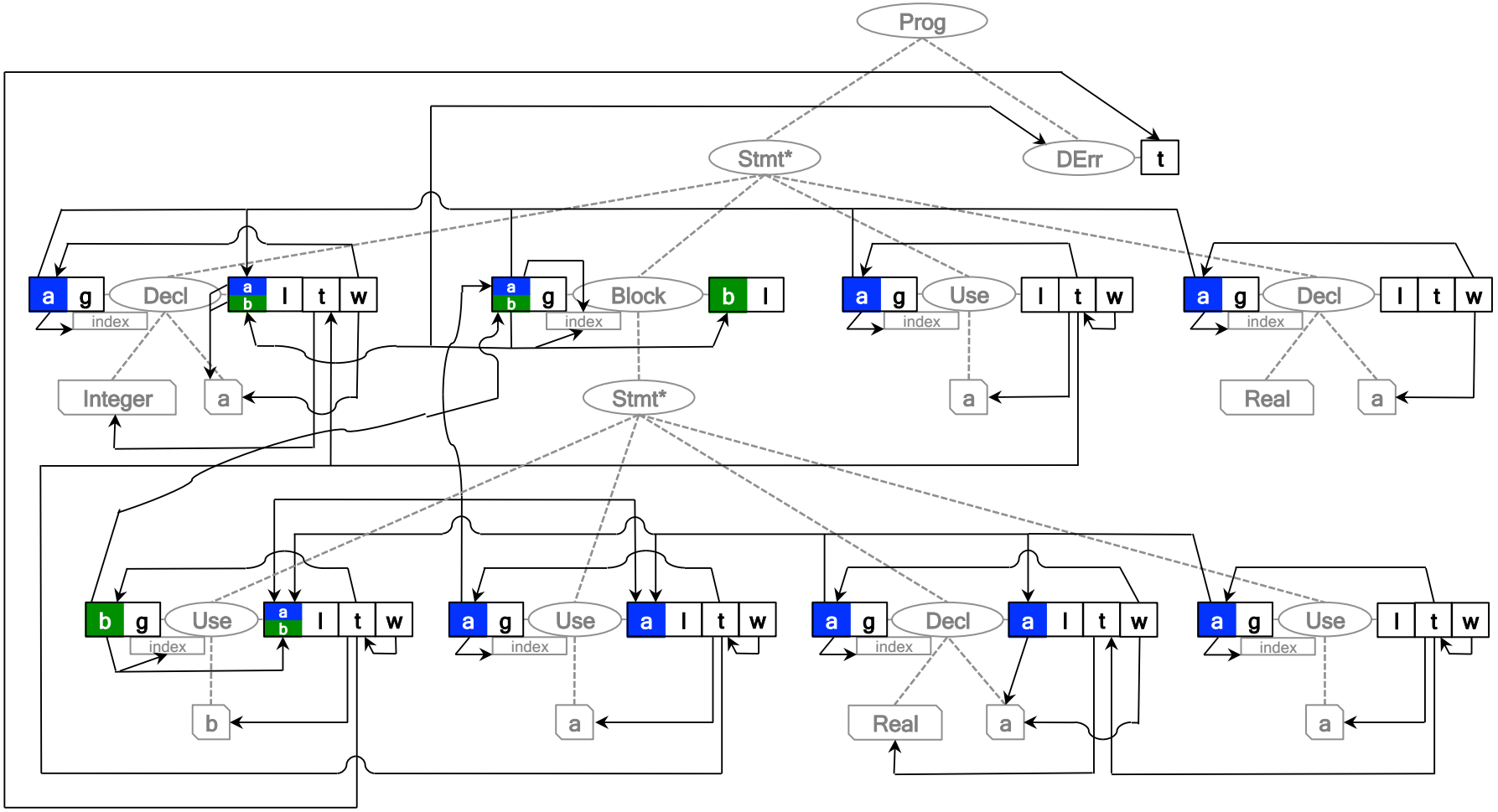
```
(ag-rule Type ; Synthesised attribute
 (Use (lambda (n) (=Type (=G-Decl n (->name n)))))
 (Decl (lambda (n) (->type n))))
```

Dynamic attribute dependency graphs



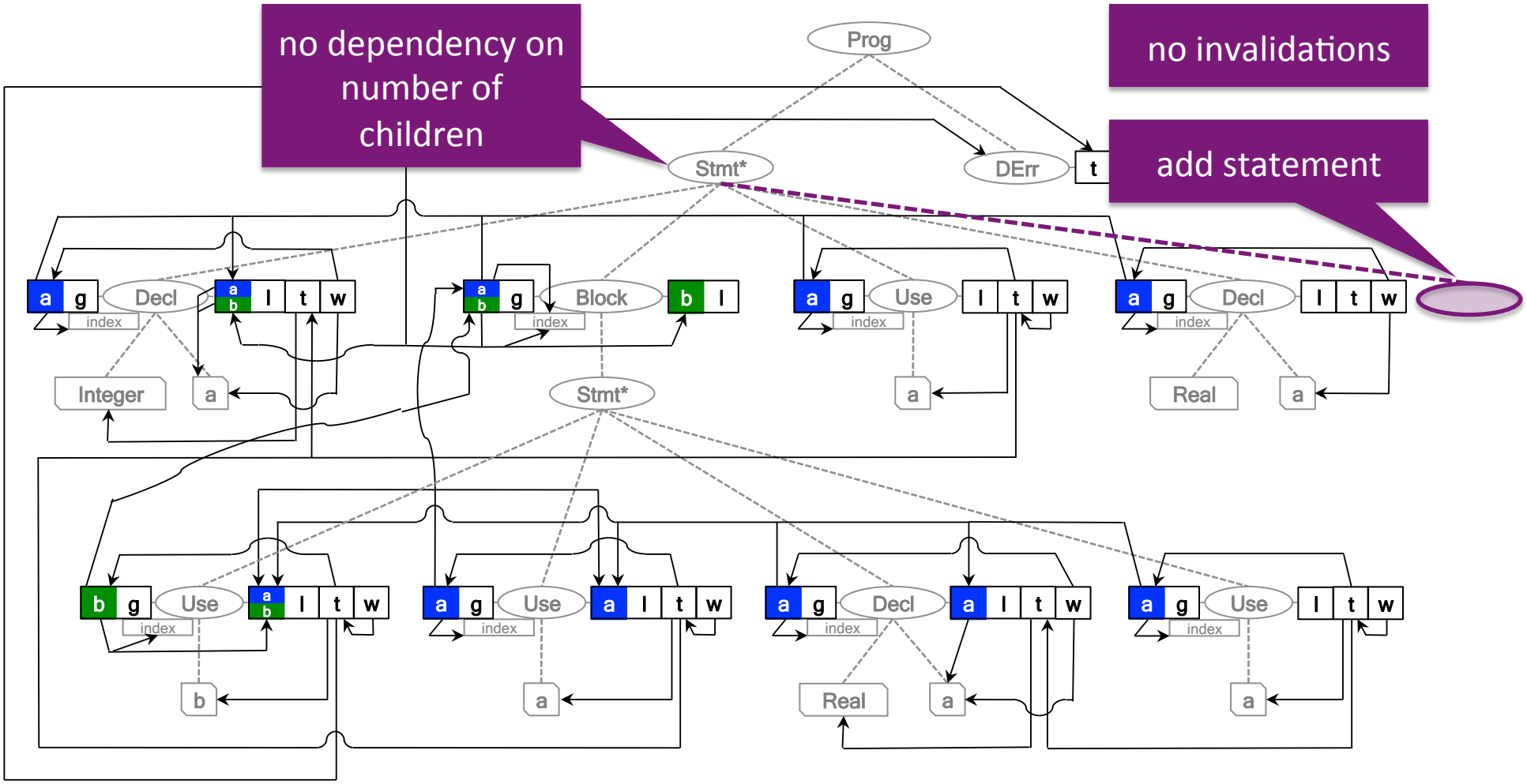
complex to achieve manually

Dynamic attribute dependency graphs

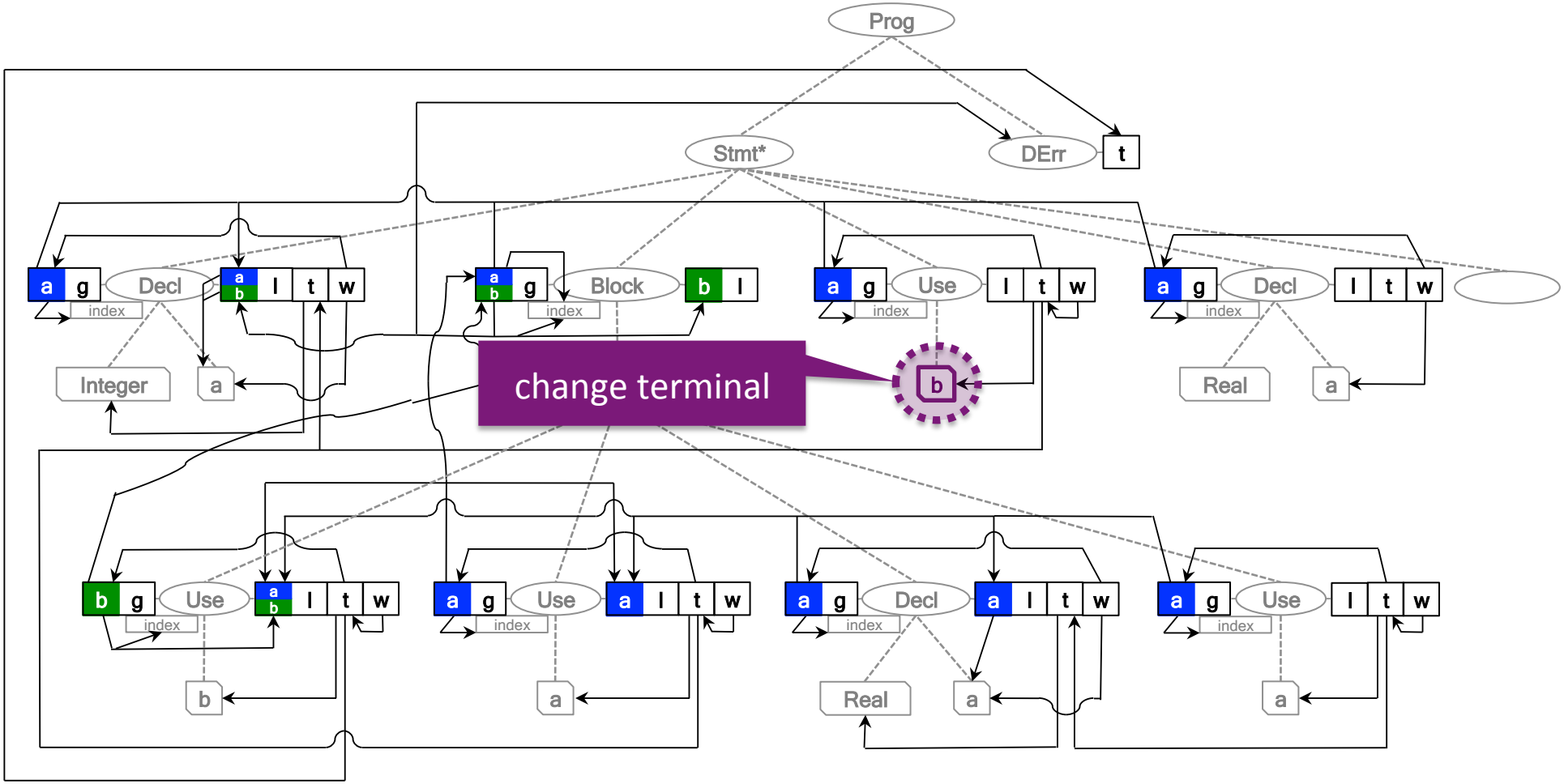


automatically in RAG-controlled rewriting

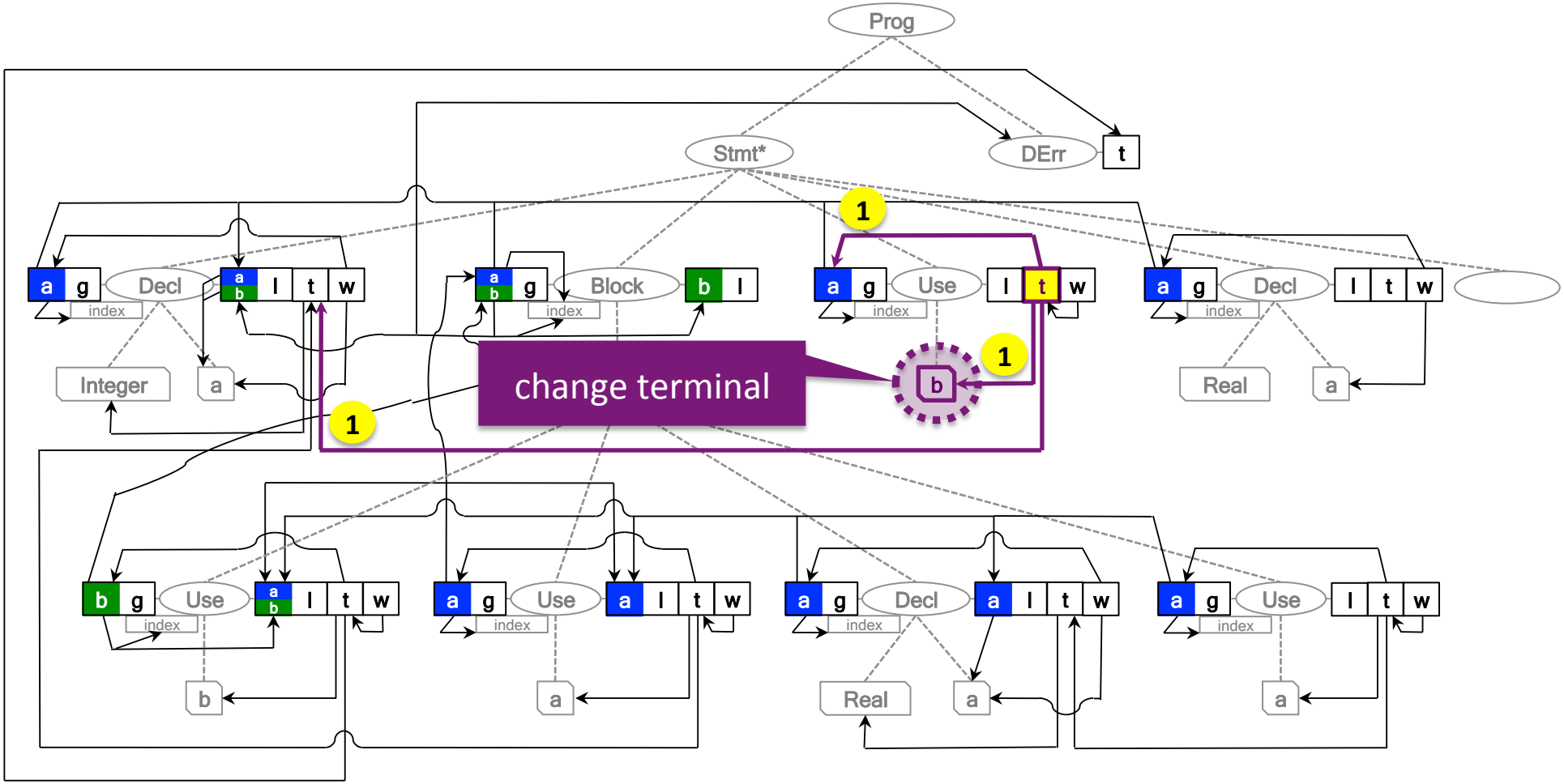
Incremental evaluation



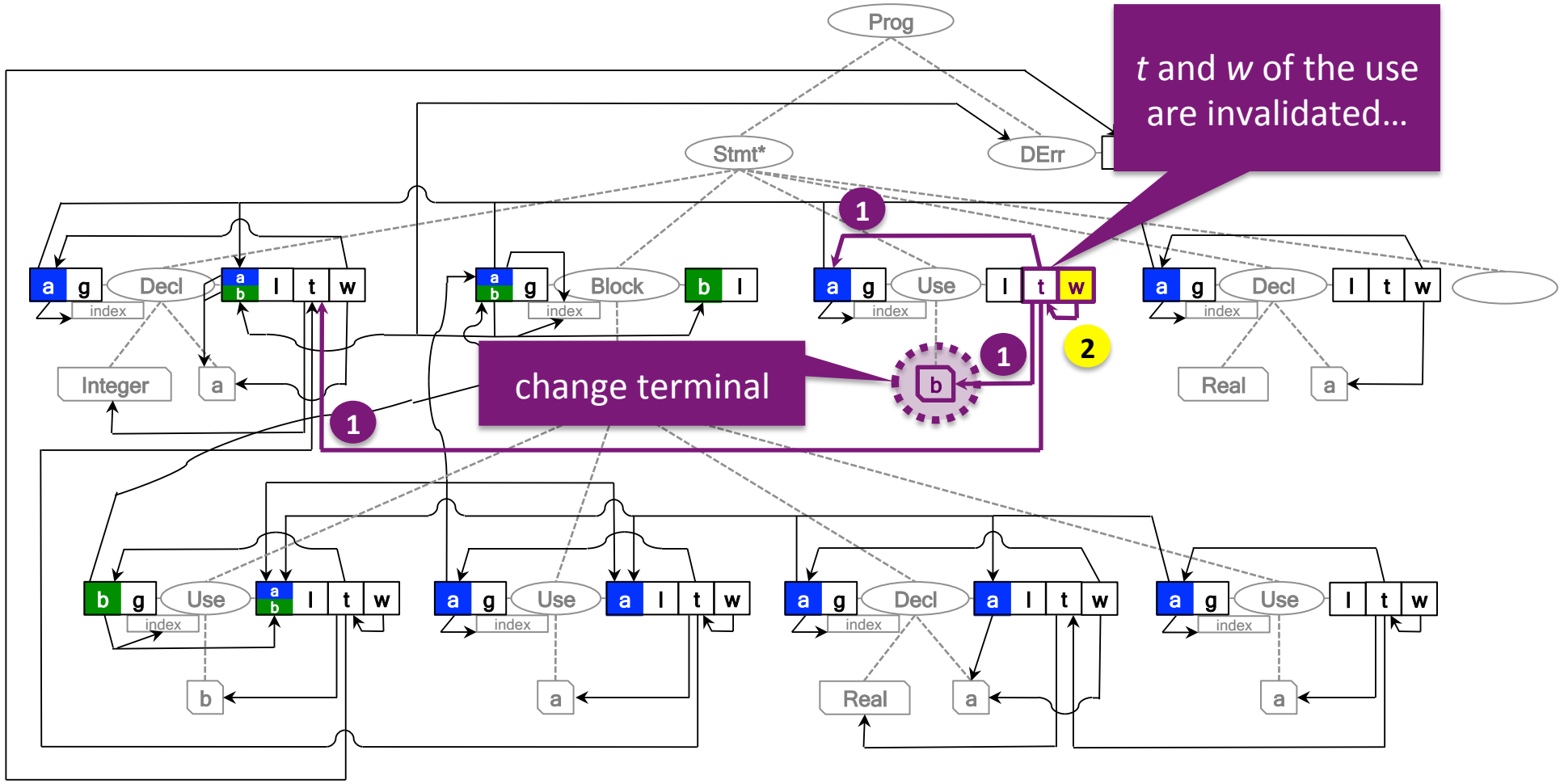
Incremental evaluation



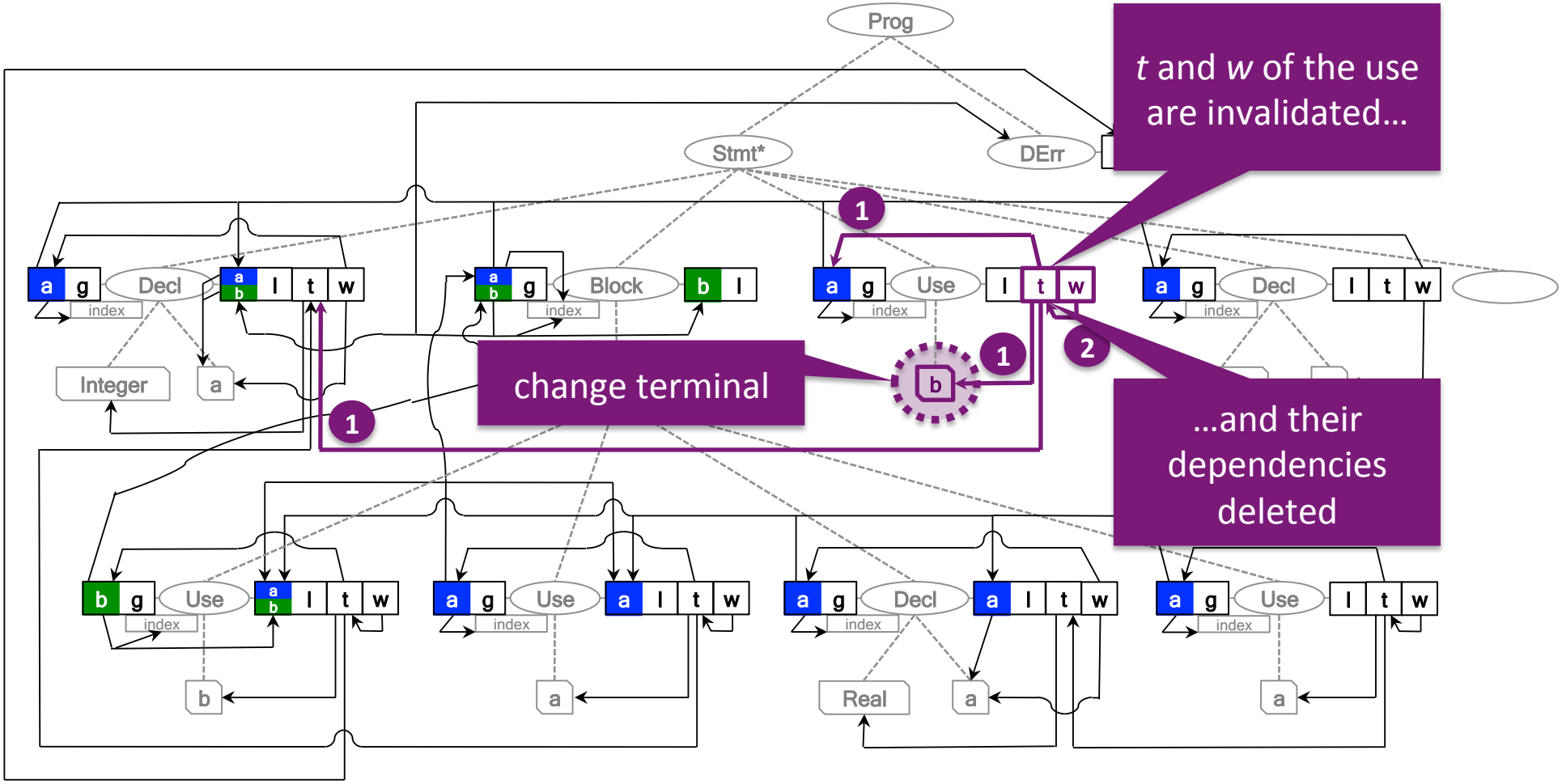
Incremental evaluation



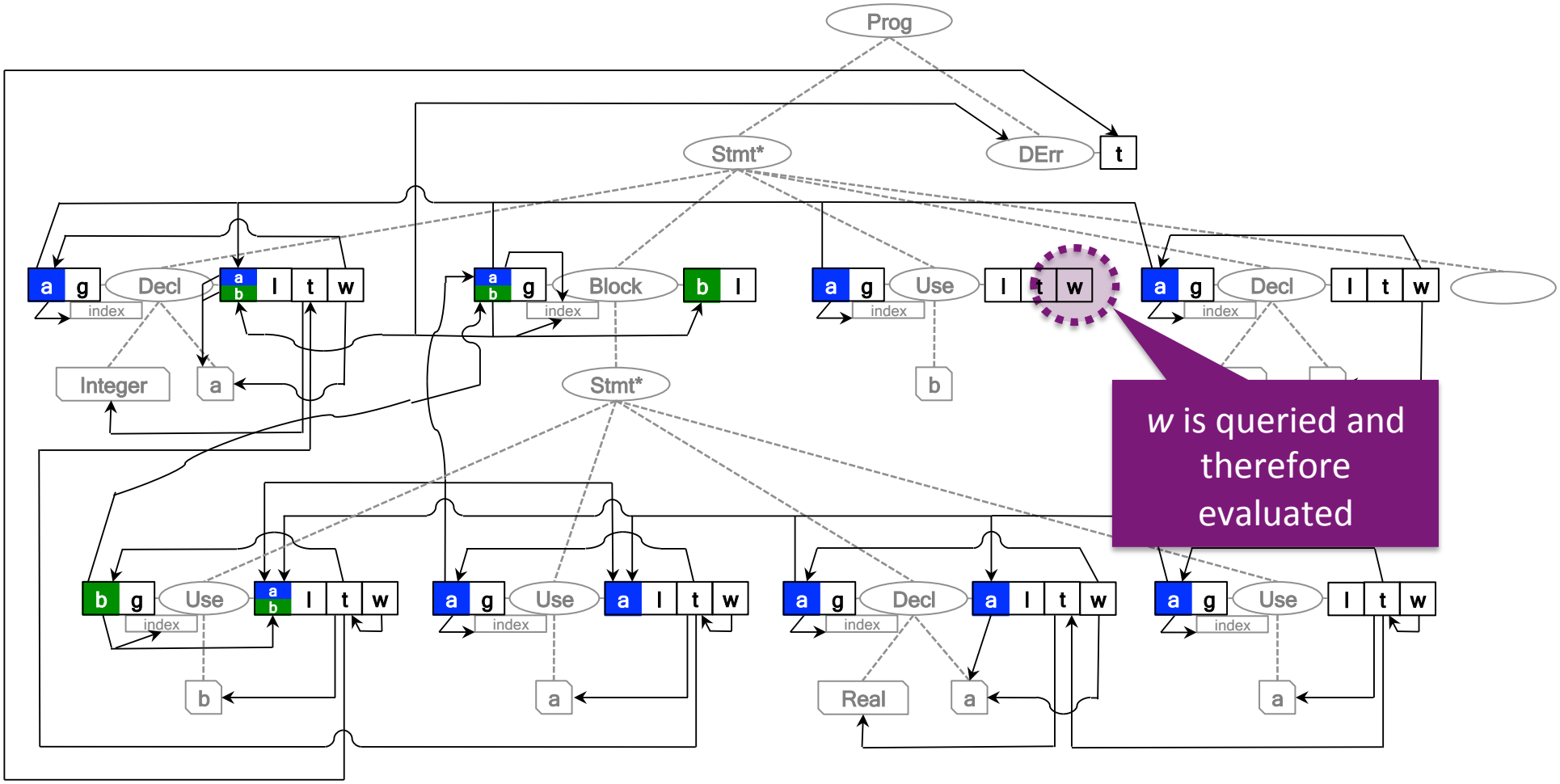
Incremental evaluation



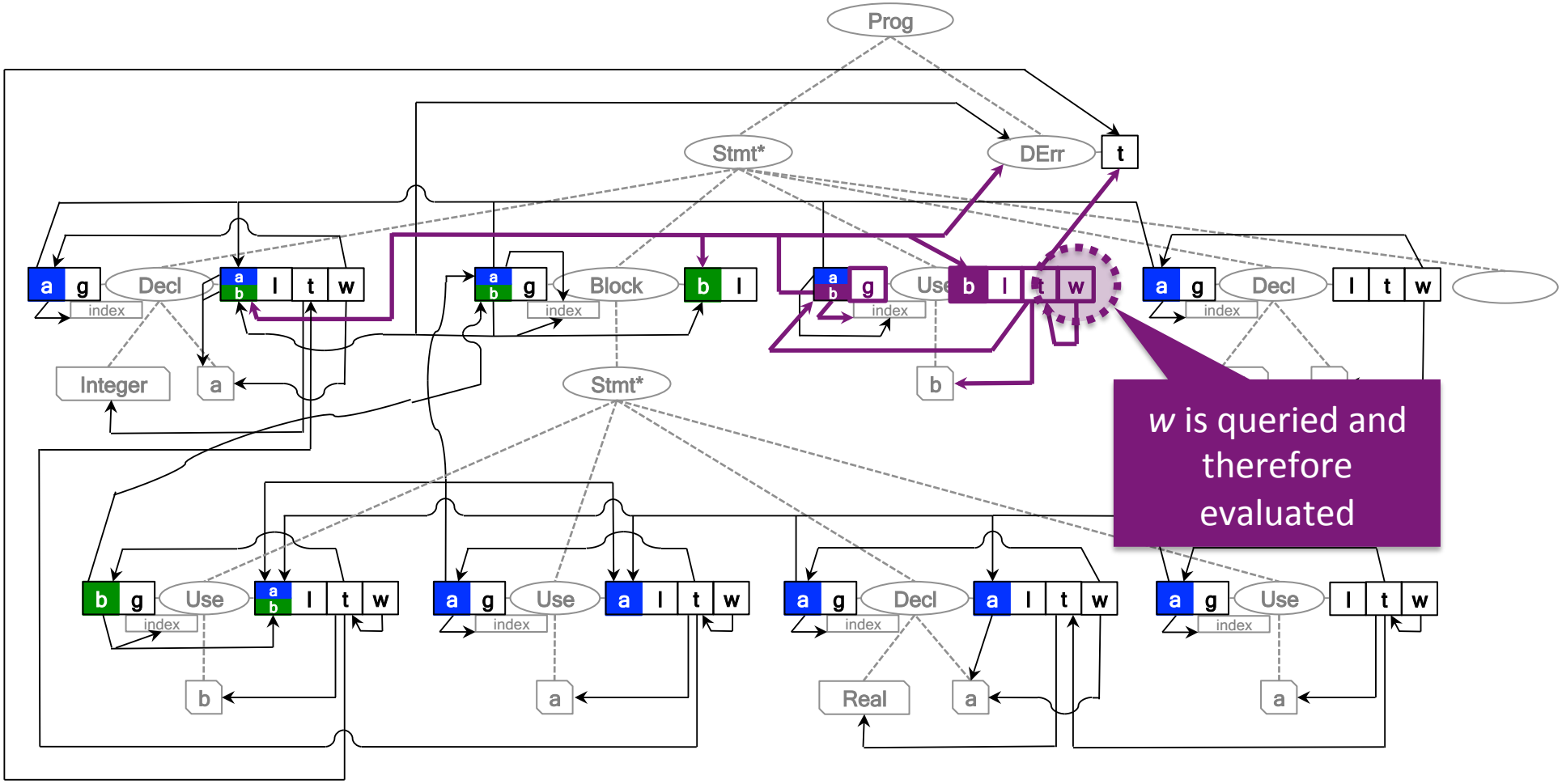
Incremental evaluation



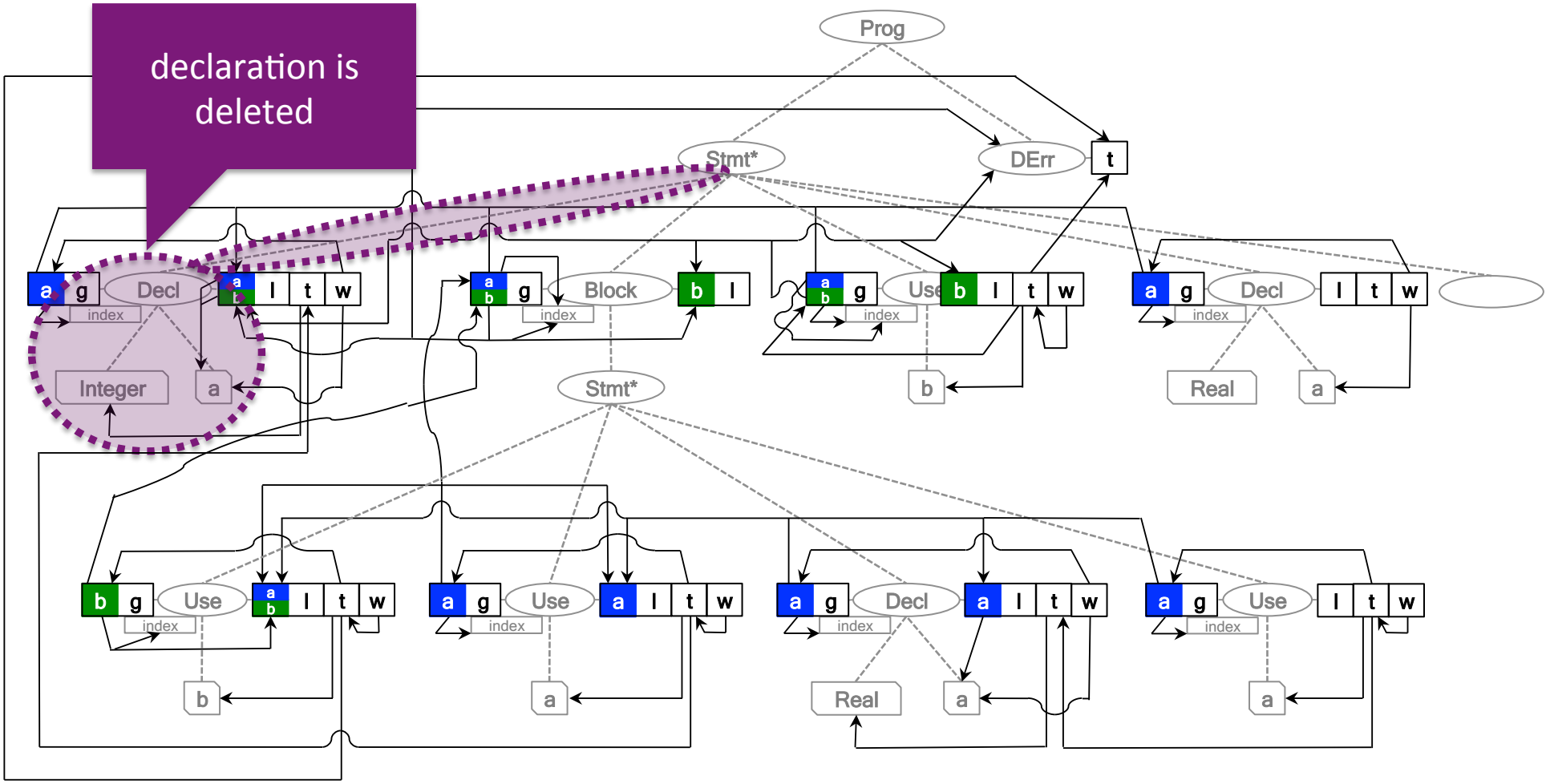
Incremental evaluation



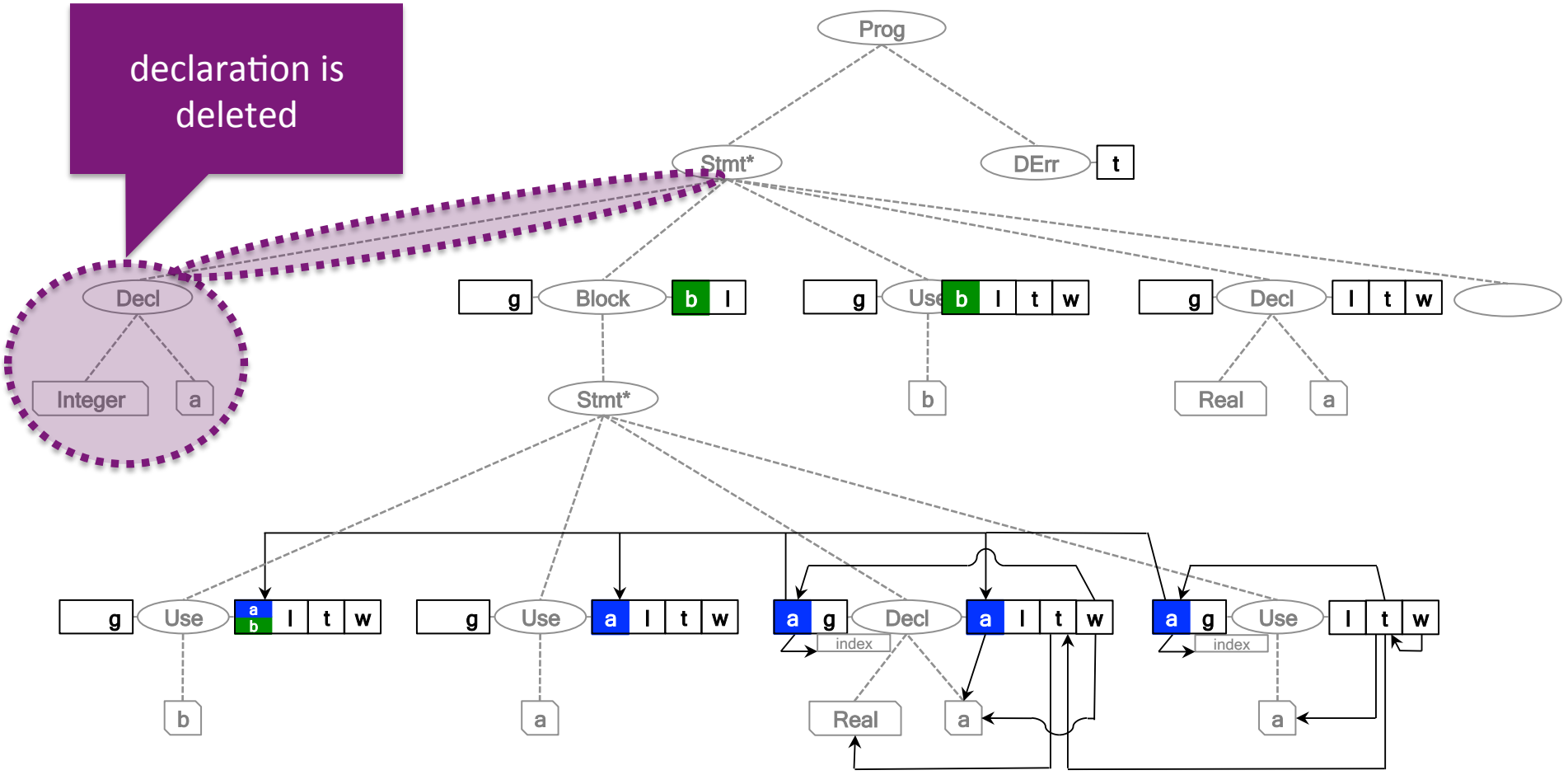
Incremental evaluation



Incremental evaluation



Incremental evaluation



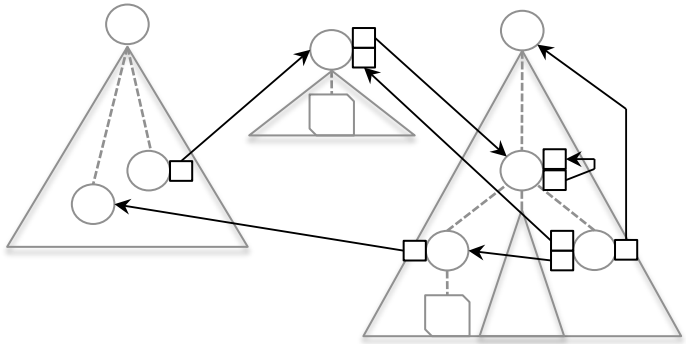
The application
How works RAG-controlled rewriting!

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*

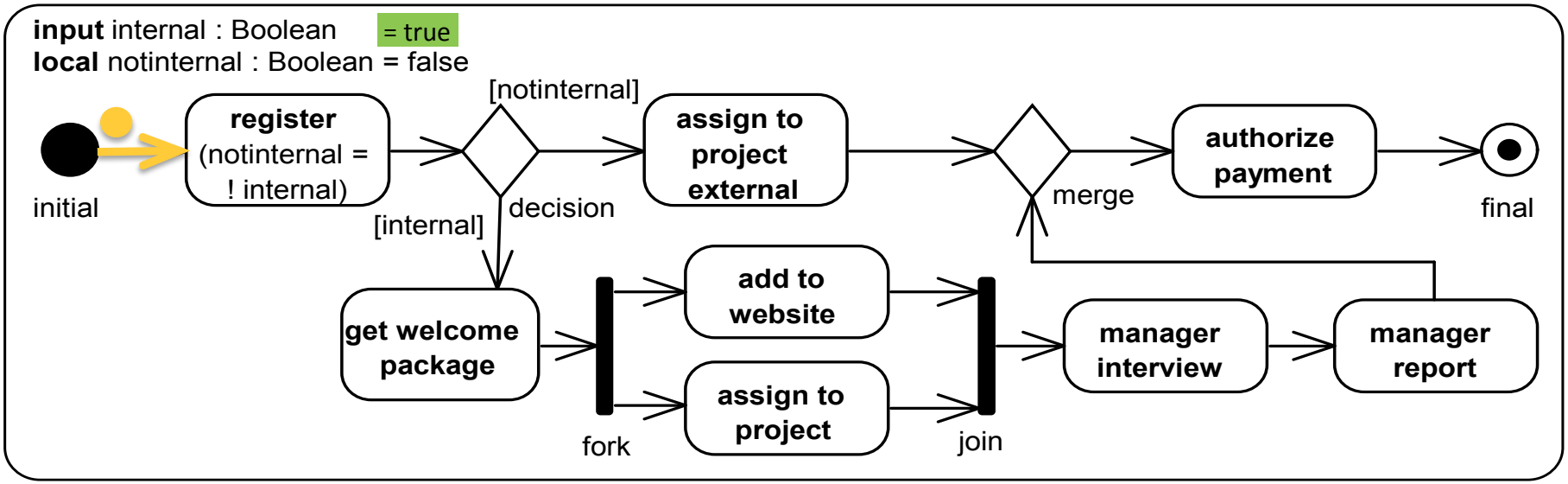
The evaluation
What is your proof of concept?

fUML Activity Diagrams[1] of *TTC 2015,*
questionnaires[1] 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

8th Transformation Tool Contest

TTC 2015

Contest Award

This document certifies that the award for

THE OVERALL QUALITY AWARD FOR THE MODEL EXECUTION CASE STUDY

has been won by

RACR

Participating team members:

CHRISTOFF BÜRGER

Location: L'Aquila, Italy

Date: 24.07.2015

Organizing Committee:

Tassilo Horn, Filip Krikava, Louis Rose

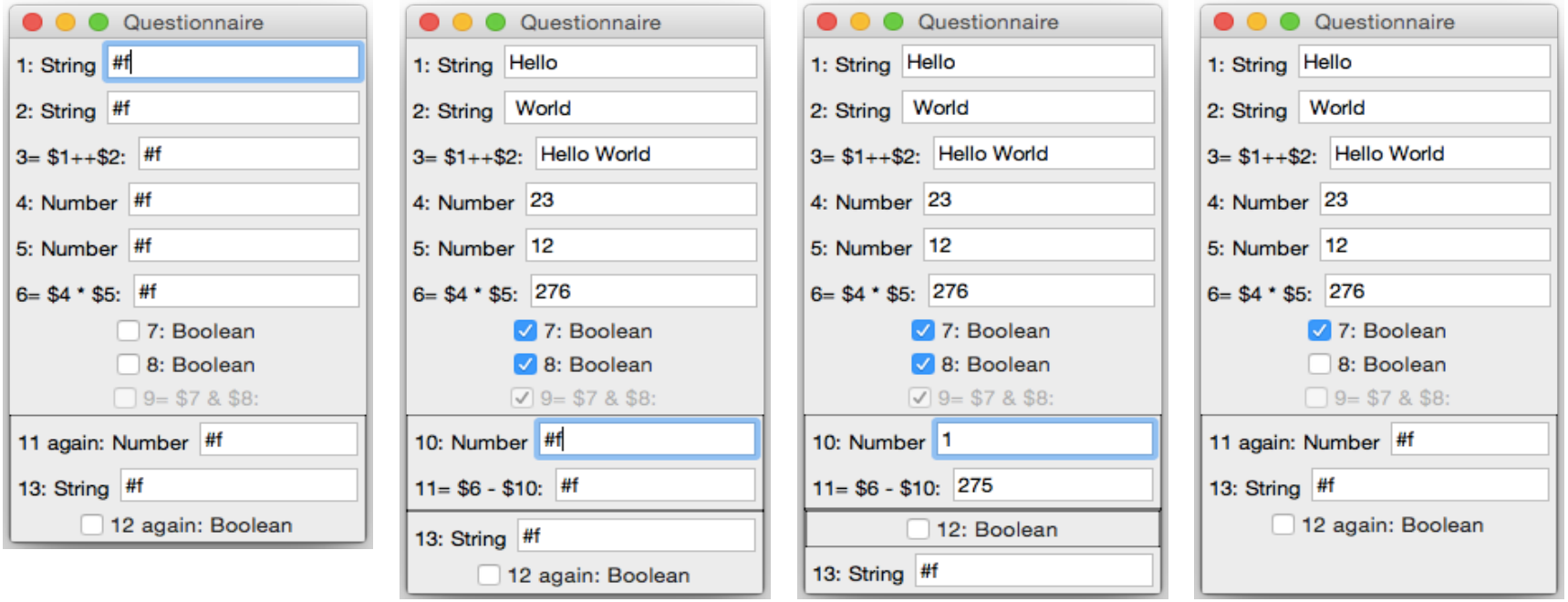
Reference

Christoff Bürger

fUML Activity Diagrams with RAG-controlled Rewriting: A RACR Solution of The TTC 2015 Model Execution Case

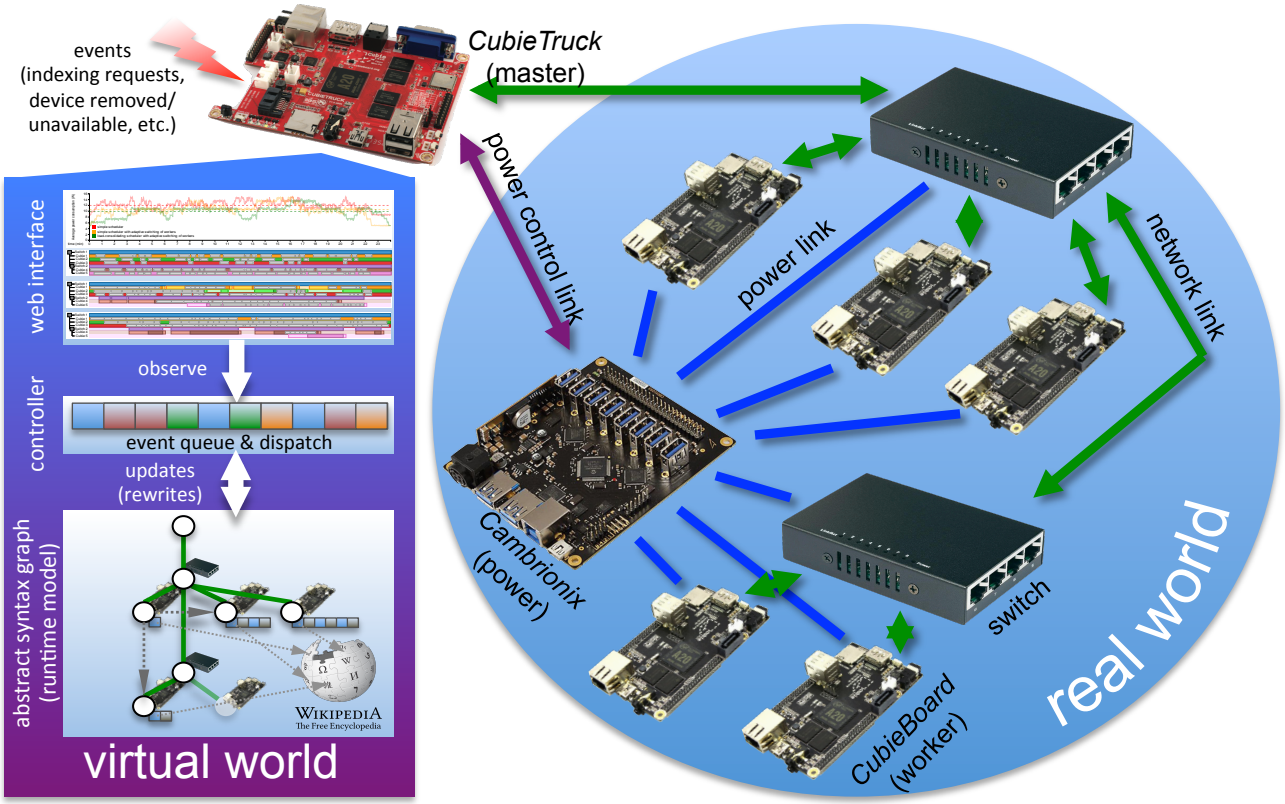
CEUR-WS.org, 2015

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

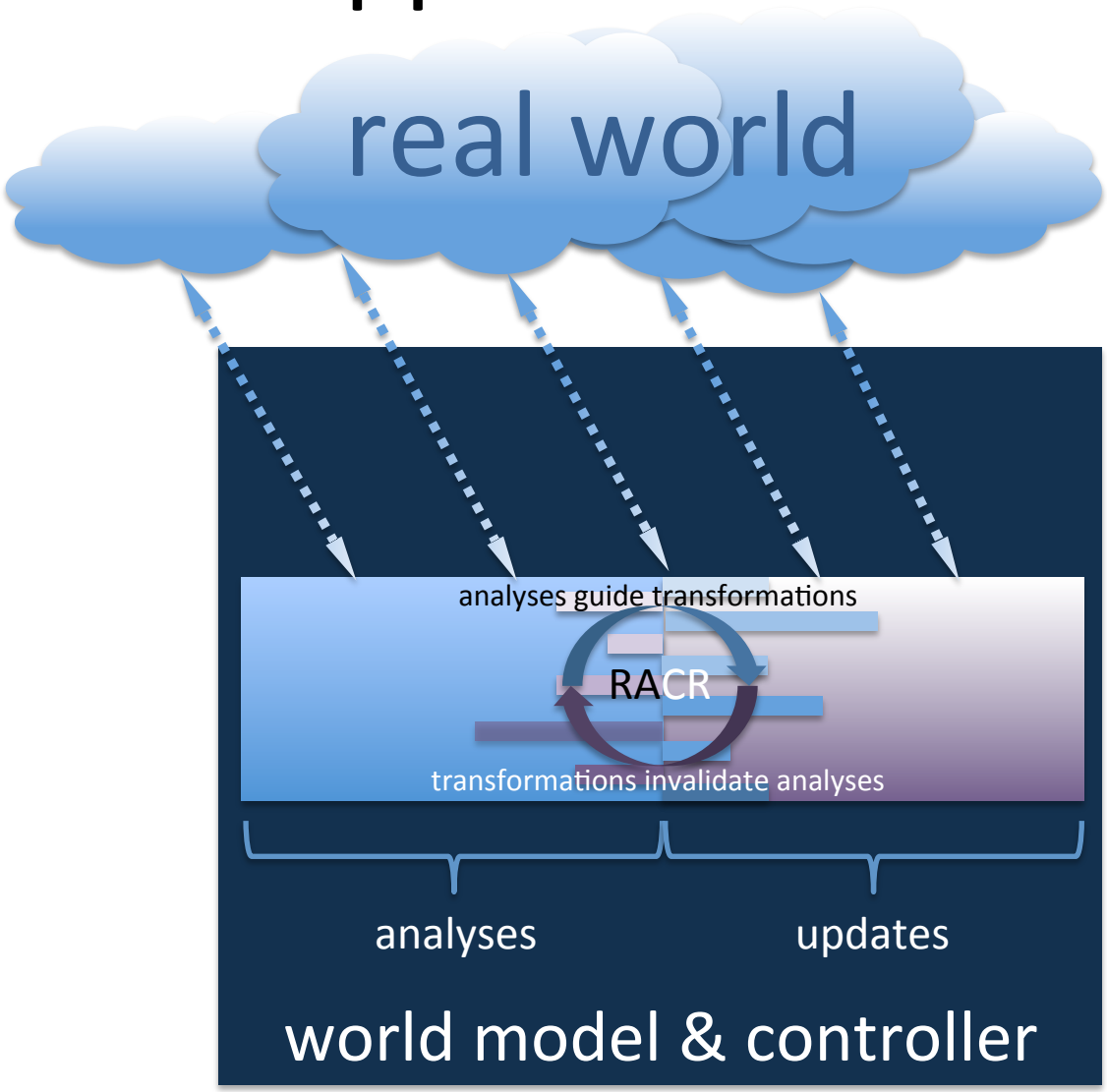
RACR solution: incremental energy efficient scheduling of indexing tasks.

The intention

What are you up to?

RAG-controlled rewriting for incremental runtime models

Intended application: runtime models



limitations of application
open research question

interactive, mutual-dependent analyses & transformations

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