Structure and Interpretation of an Aspect Language for Datatype

Karl Lieberherr

- fix 27
- semantics: go everywhere and collect ogs.
- then apply visitors
- general strats: exponentially many paths

2 Lectures

- The motivation and theory behind “Aspect Language for Datatype”.
  - datatypes and class graphs
  - Semantics (// A B) (navig-object graphs.*)
  - Visitors and type checking
- Interpreter implementation

Motivation

- Build on foundations that Matthias presented.
- Connections to templates: stressing the importance of structural recursion.
- Not only an interpreter but also a compiler (works, because traversals are sufficiently simple).
- Very useful application of foundations that is in itself a foundation.
- Demonstration that simple languages can be full of surprises.

Homework

- Simple aspect-oriented language.
- Leads to a radically different way of programming: programming without knowing details of data structures. Write programs for a family of related data structures.
- Northeastern SAIC project ca. 1990.

Homework evolution

- Initial motivation: make EOPL datatype style programming easier by adding a traverse function.
- Visitors written in full Scheme: AdaptiveScheme = Scheme + EOPL datatype + traversal strategies + visitors.
- You get a simplified form (thanks Matthias).

Interpretation

- Interpret a traversal on an object tree.
- (join (//A B) (//B C)): starting at an A-node, traverse entire object tree, return C-nodes that are contained in B-nodes that are in turn contained in A-nodes.
- Not interesting enough. Can meta information about object trees make it more interesting?

Interpretation with meta information

- Use a graph to express meta information.
- Many applications:
  - data type / data trees
  - class graph / object trees
  - schema / documents (XML)
  - programs / execution trees

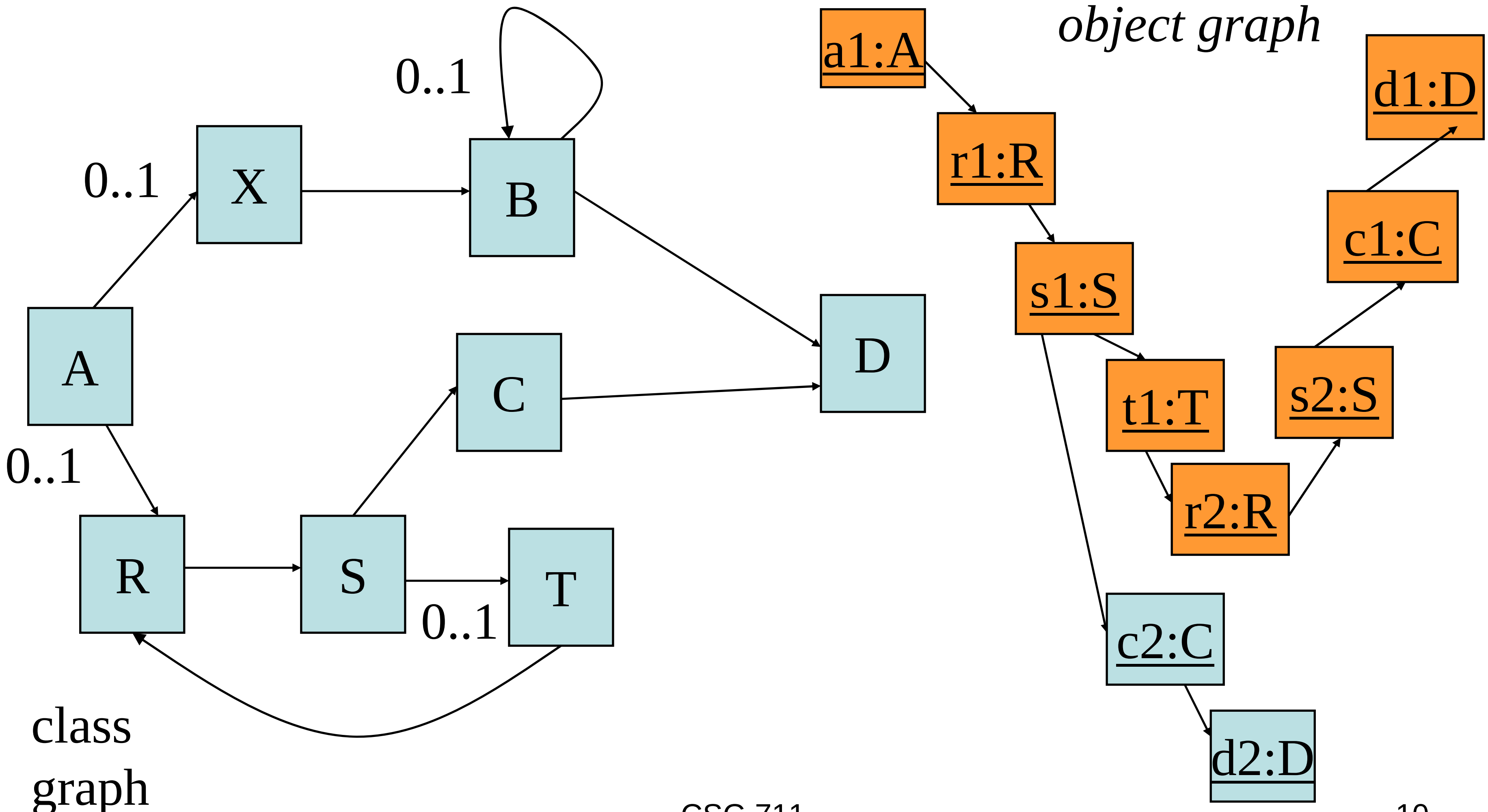
Class graphs (simplified UML class diagrams)

- nodes and edges
- nodes: concrete and abstract
- edges: has-a (triples) and is-a (pairs)
- concrete nodes: no incoming is-a
- supports inheritance
- flat: a class graph is flat if no abstract node has an outgoing has-a edge

Example B2

strategy

`A//T//D`



Plan

- (M s cg og) = ?
  - (M1(M2 s cg) og) = ?
  - og satisfies cg!
- Not only traverse!
- (Mv s cg og V)
  - (Mv1 (M1 (M2 s cg) og) V)
  - visitor V: before / after applications to node / edge. Local storage. Visitor functions are activated by traversal.

Sample visitor

```
(visitor PersonCountVisitor
  0 // initial value
  PersonCountVisitor // return
  before (host Person)
    (+ PersonCountVisitor 1)
)
```

count all persons waiting at any bus stop on a bus route

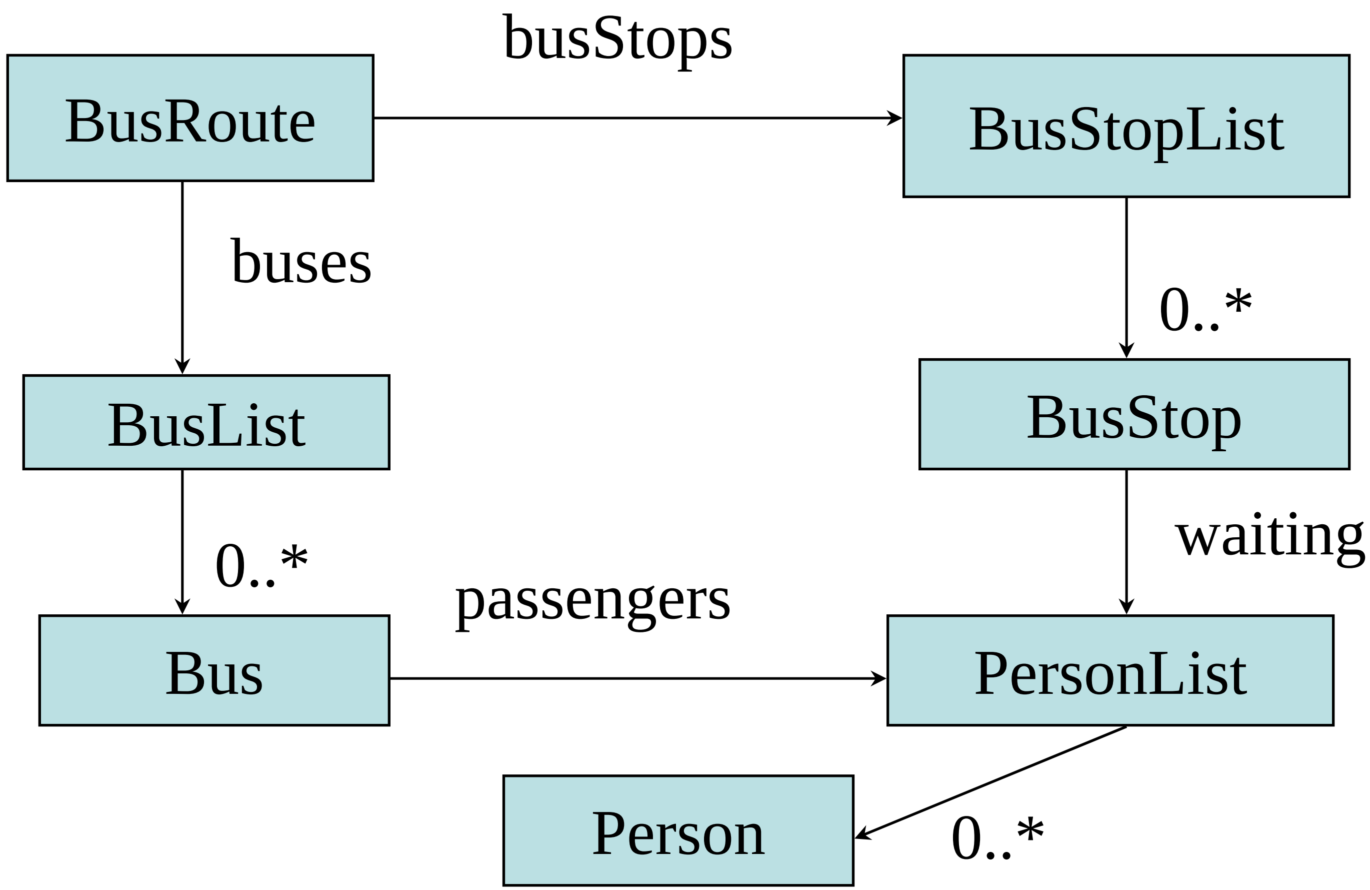
Example

- (Mv s cg og PersonCountVisitor)
- cg : class graph for bus routes
- og: object graph for bus routes
- s = (join (// BusRoute BusStop)

  (// BusStop Person))

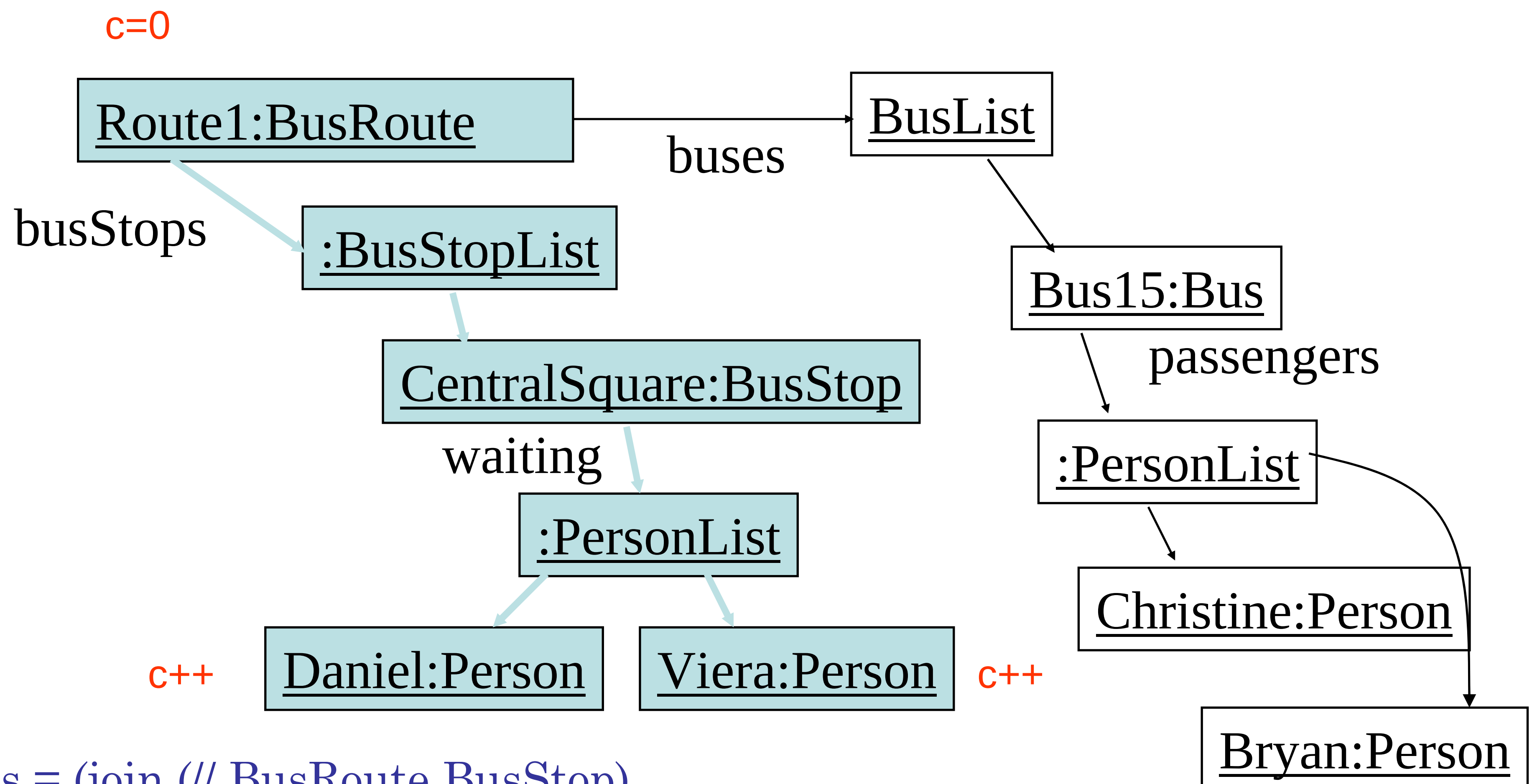
Class Graph

count all persons waiting at any bus stop on a bus route



Object Graph

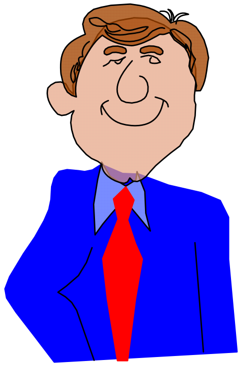
(Mv s cg og PersonCountVisitor) = ??



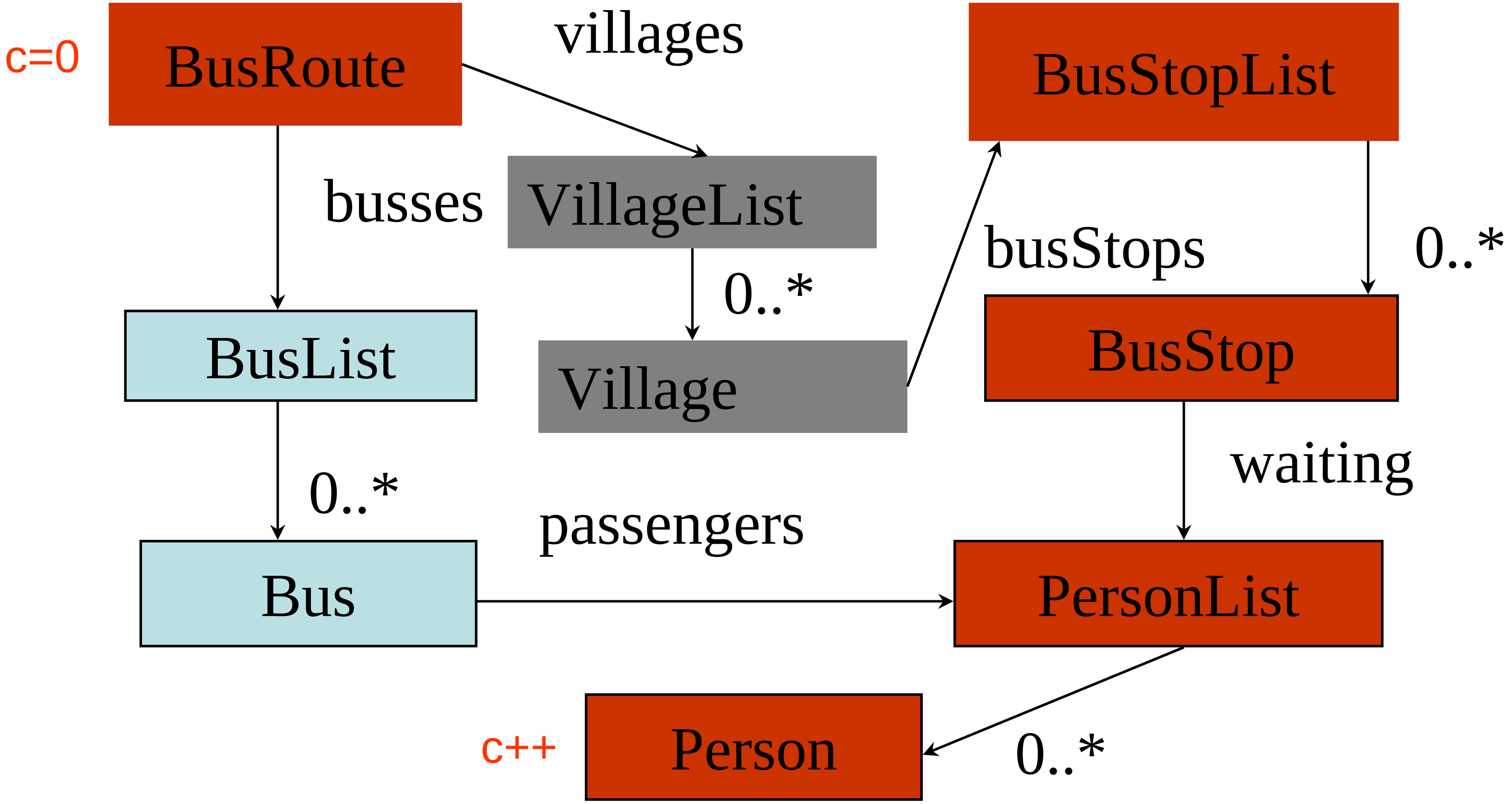
s = (join (// BusRoute BusStop)
(// BusStop Person))

count all persons waiting at any bus stop on a bus route

Robustness



s = BusRoute // BusStop // Person



Aspects

- Aspects as program enhancers
- Here we enhance traversal programs with before and after advice defined in aspects called visitors
- General AOP enhances any kind of program
- This is a special case with good software engineering properties

Develop a sequence of semantics

- (M s cg og) = ?
  - og satisfies cg!
- s, cg, og: are graphs. Graphs are relations. Use terminology of relations.
- Restrict to s = (// A B).

Object level semantics

- (M s cg og), where s = (// A B).
- The key is to find a set FIRST(A,B) of edges such that e in FIRST(A, B) iff it is possible for an object of class A to reach an object of type B by a path beginning with an edge e.
- (M s cg og) is the FIRST(A,B) sets.

Homework class graphs

A CG is: (DD+)

HW class graph to class graph transformation:
TypeName -- abstract class
AlternativeName – concrete class
(AlternativeName, FieldName, TypeName) – has-a
(AlternativeName, TypeName) – is-a

A DD is:

(datatype TypeName Alternative+)

An Alternative is:

(AlternativeName (FieldName TypeName)+)

Homework class graphs

CD = PL(DD).

DD = "(datatype" TypeName L(Alternative) ")".

Alternative = "(" AlternativeName L(TypedField) ")".

TypedField = "(" FieldName TypeName ")".

FieldName = Ident.

TypeName = Ident.

AlternativeName = Ident.

L(S) ~ {S}.

PL(S) ~ "(" {S} ")".

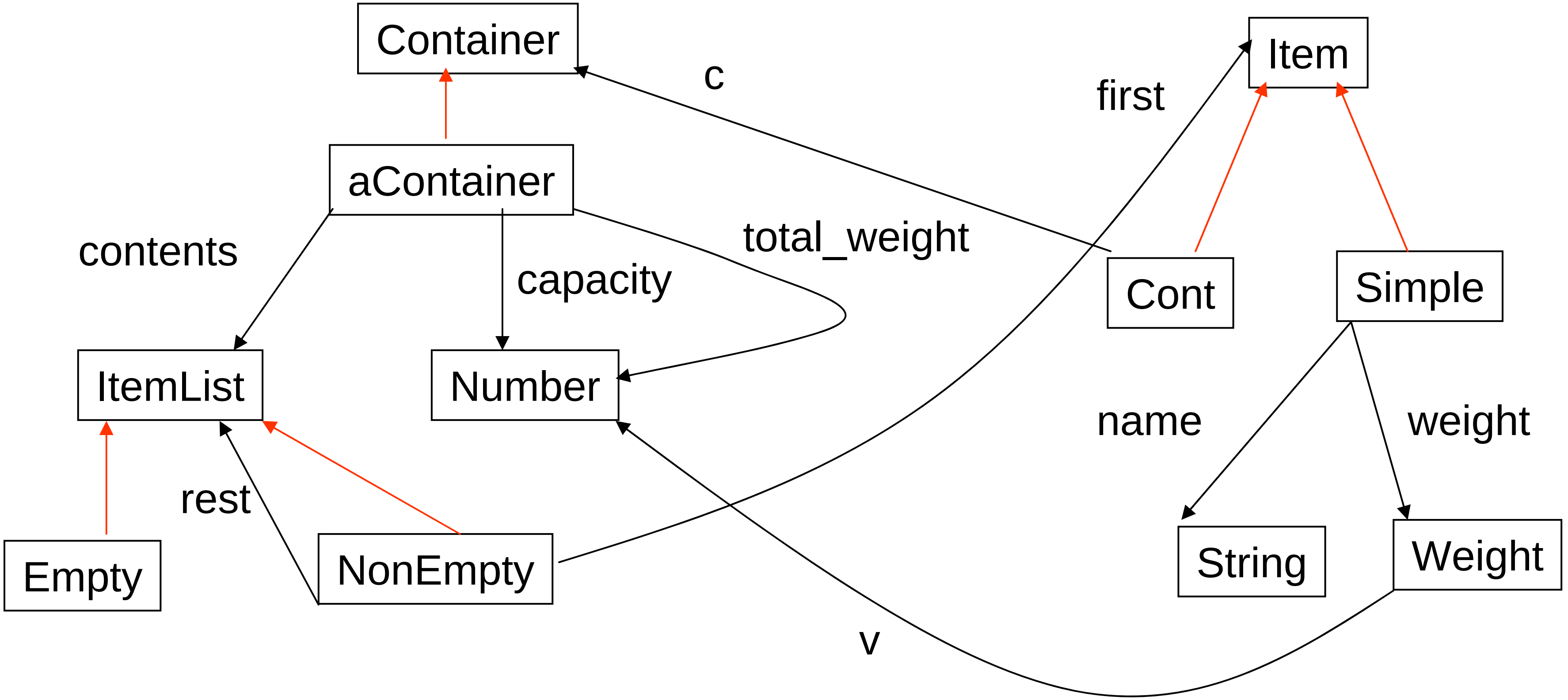
Class graph example

```
(datatype Container
  (a_Container (contents ItemList)
               (capacity Number)
               (total_weight Number)))
(datatype Item
  (Cont (c Container))
  (Simple (name String) (weight Weight)))

(datatype Weight
  (a_Weight (v Number)))

(datatype ItemList
  (Empty)
  (NonEmpty (first Item) (rest ItemList))))
```

traversal strategy:
(// a_Container a_Weight)

HW class graph to class graph transformation:
TypeName   -- abstract class
AlternativeName – concrete class
(AlternativeName, FieldName, TypeName) – has-a
(AlternativeName, TypeName) – is-a

As traditional class graph



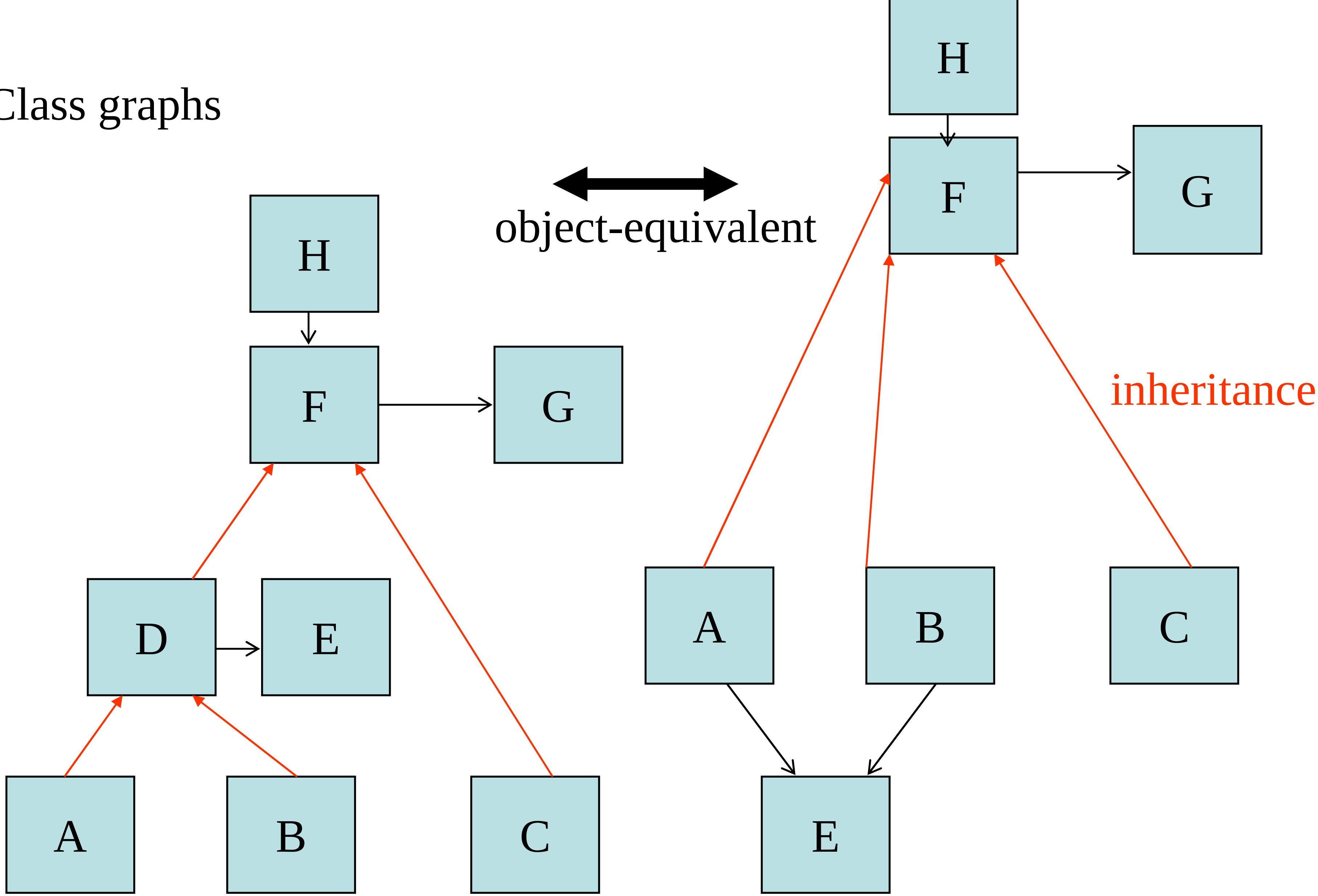
Another class graph example

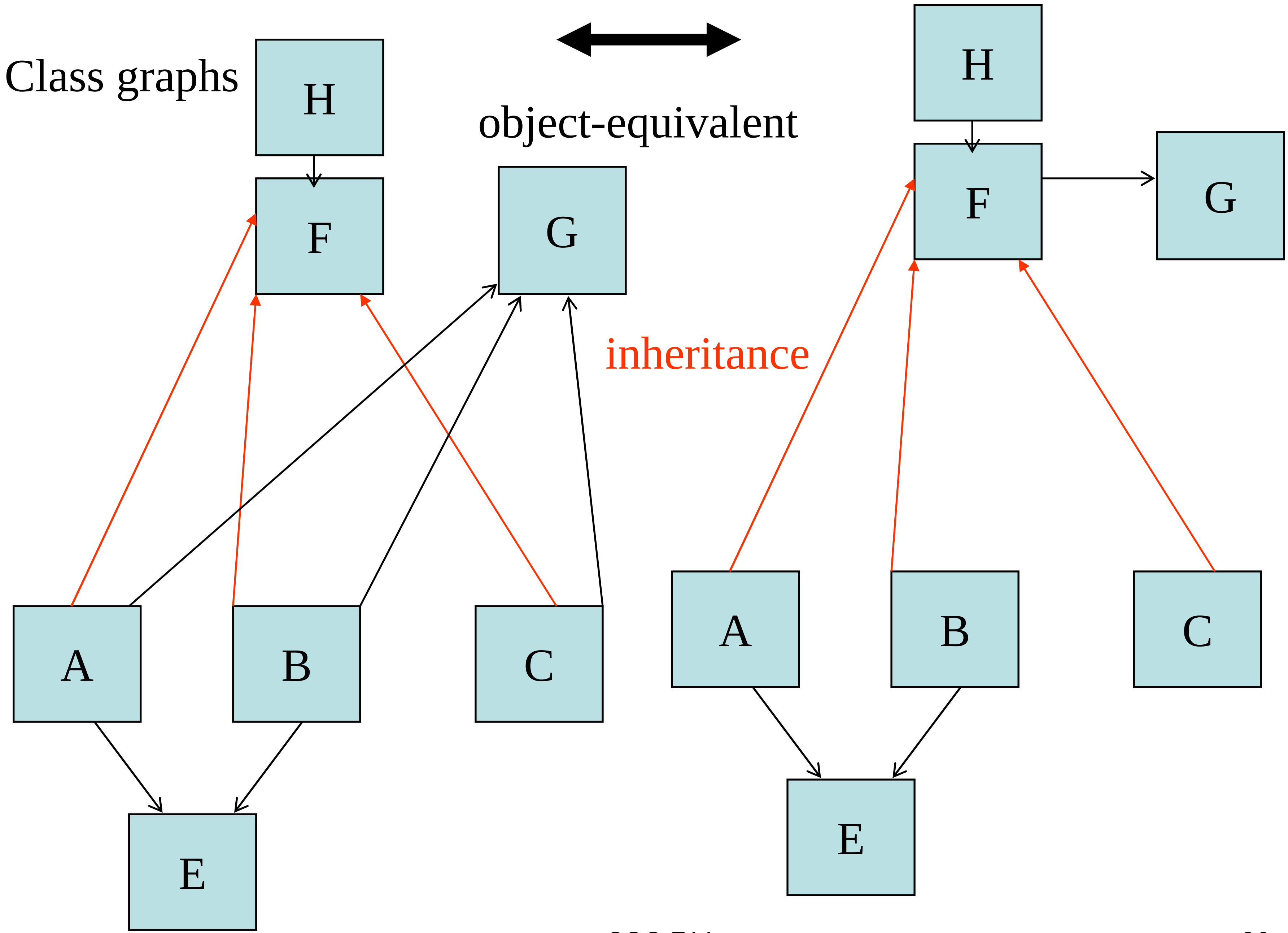
(datatype P (CP (q Q)))

(datatype Q (CQ (p P)))

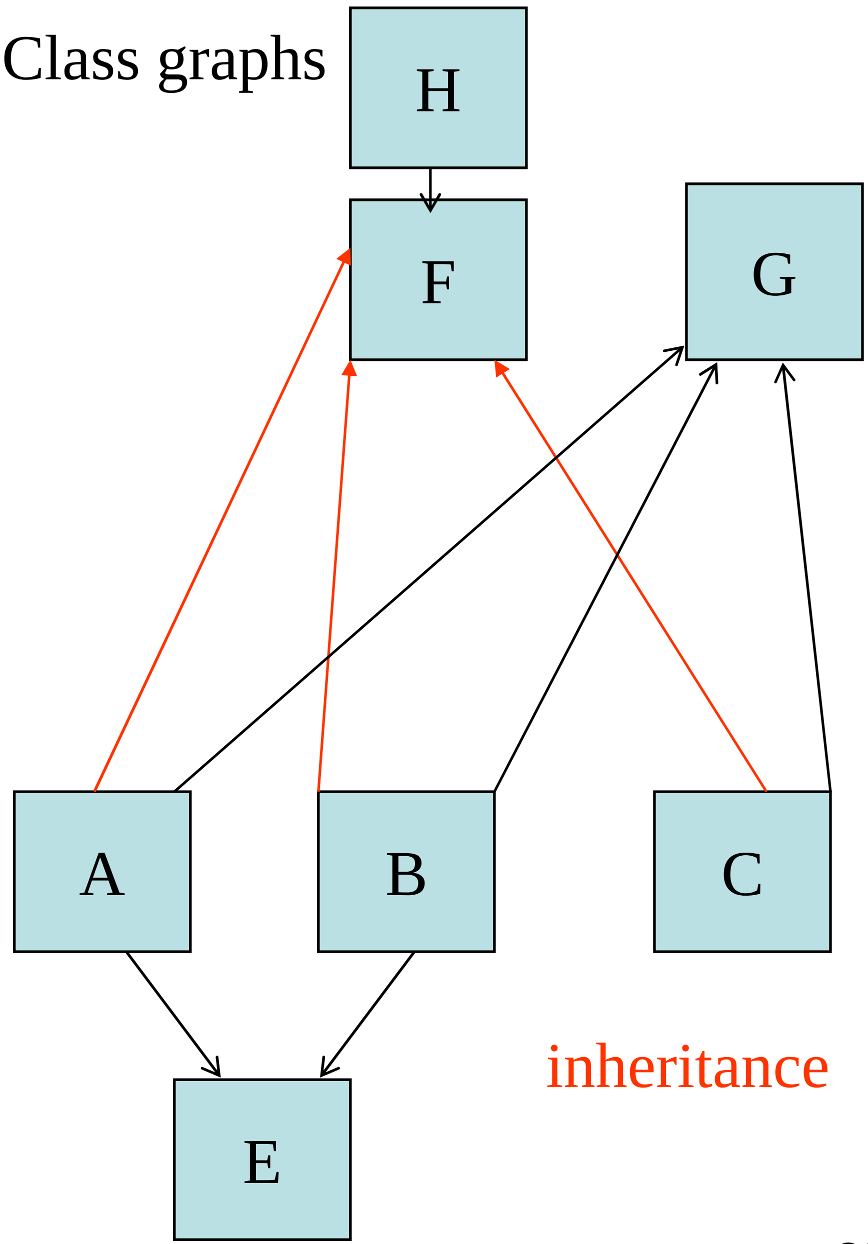
Because we only allow trees for object graphs, we should disallow such class graphs? P and Q are useless.

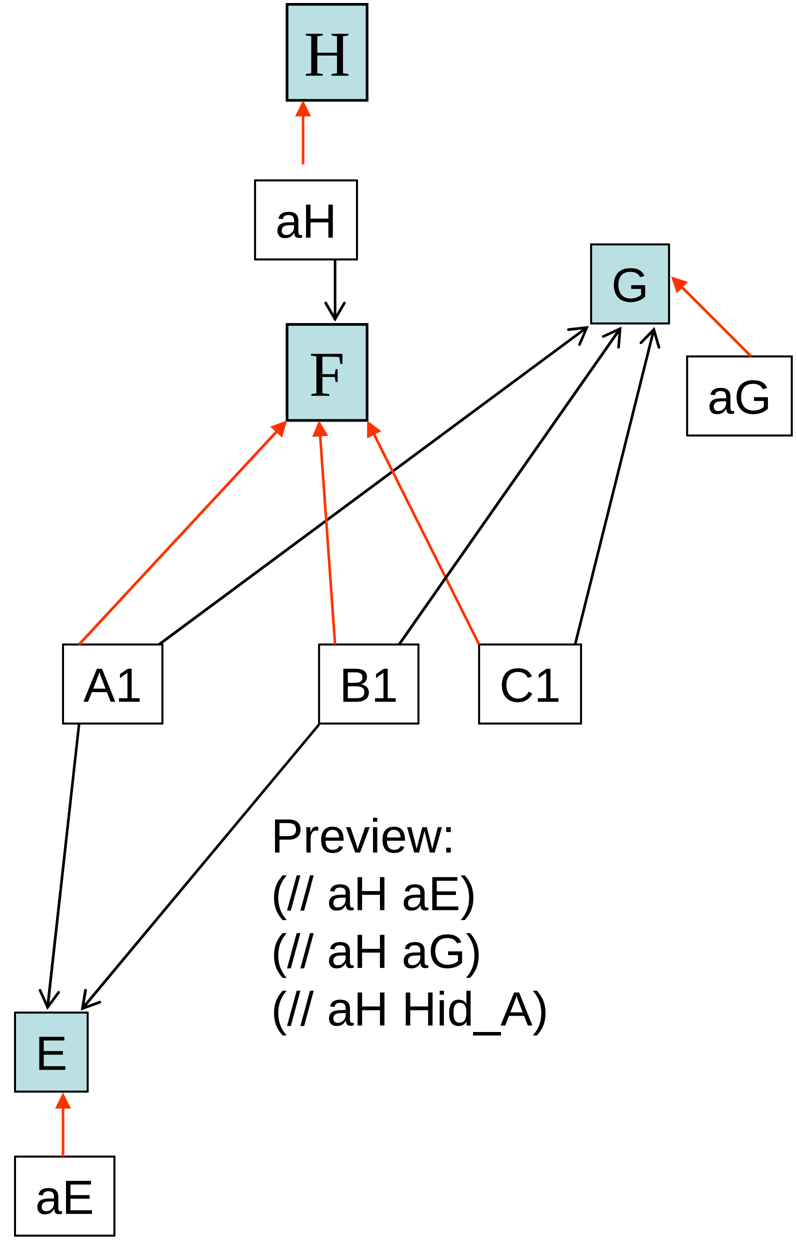
Class graphs




Class graphs
object-equivalent
inheritance
H
F
G
A
B
C
E
H
F
G
A
B
C
E

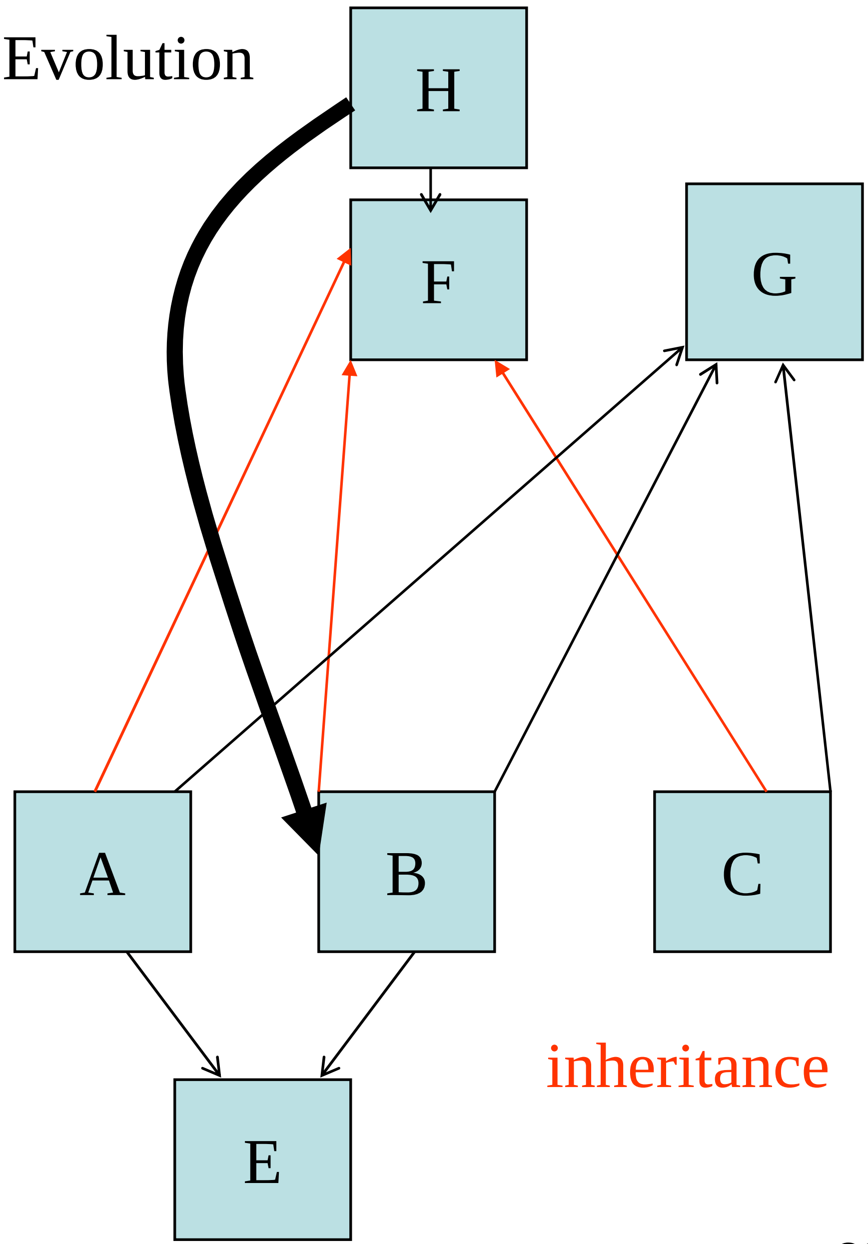
Class graphs

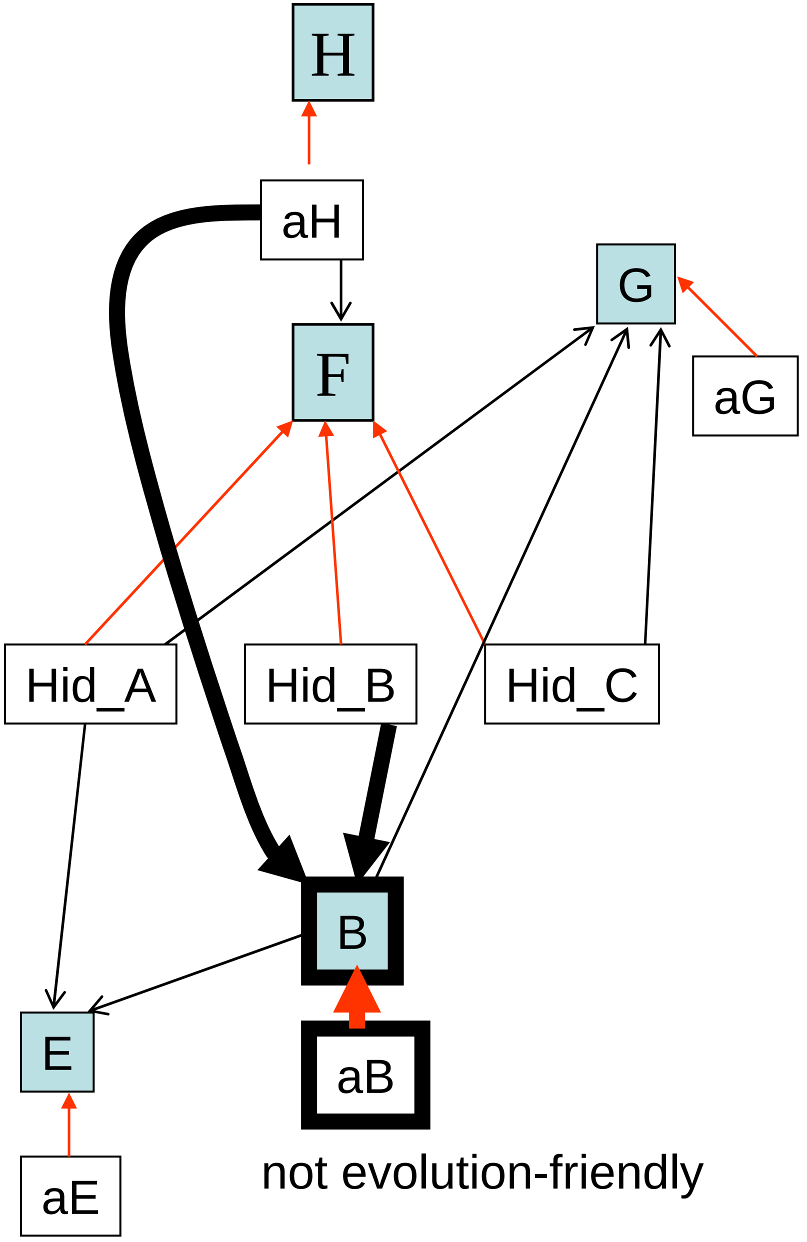




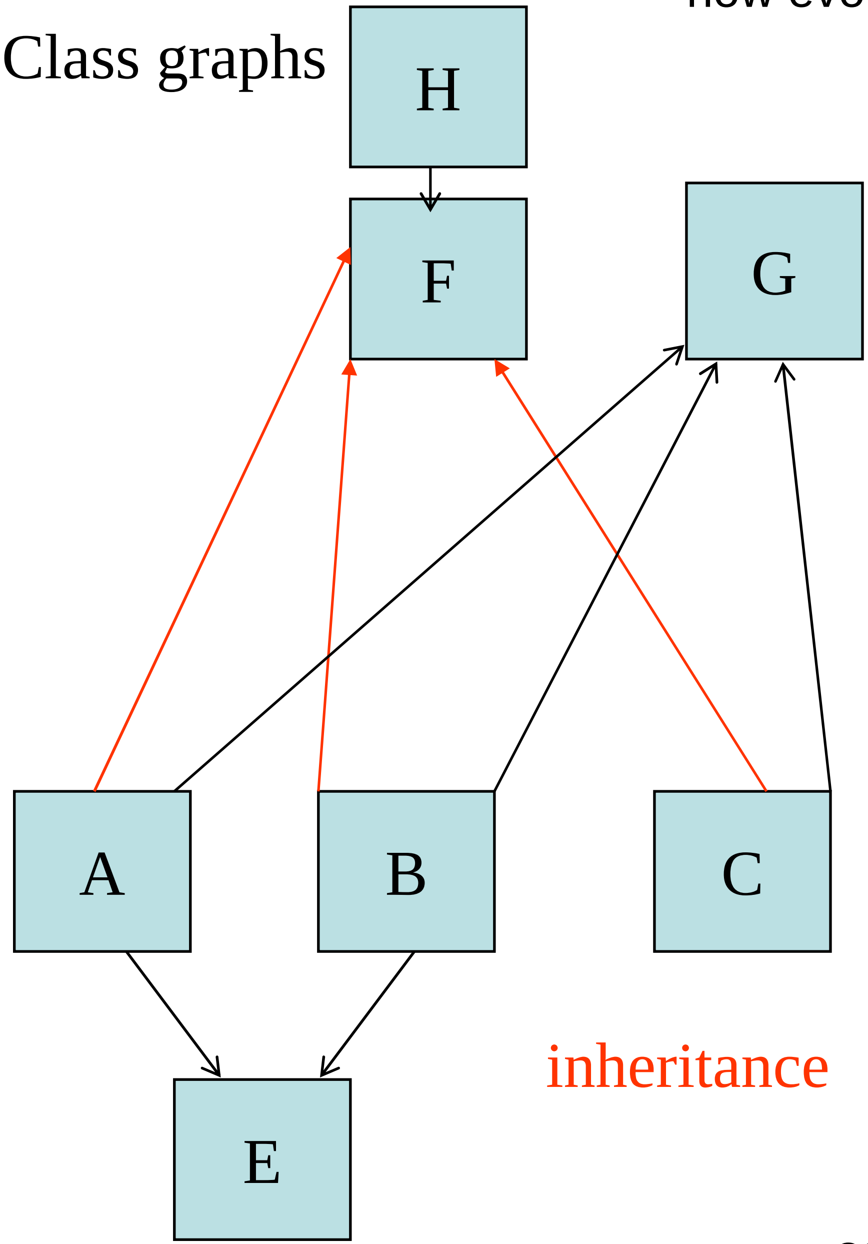
Preview:
(// aH aE)
(// aH aG)
(// aH Hid_A)

Evolution

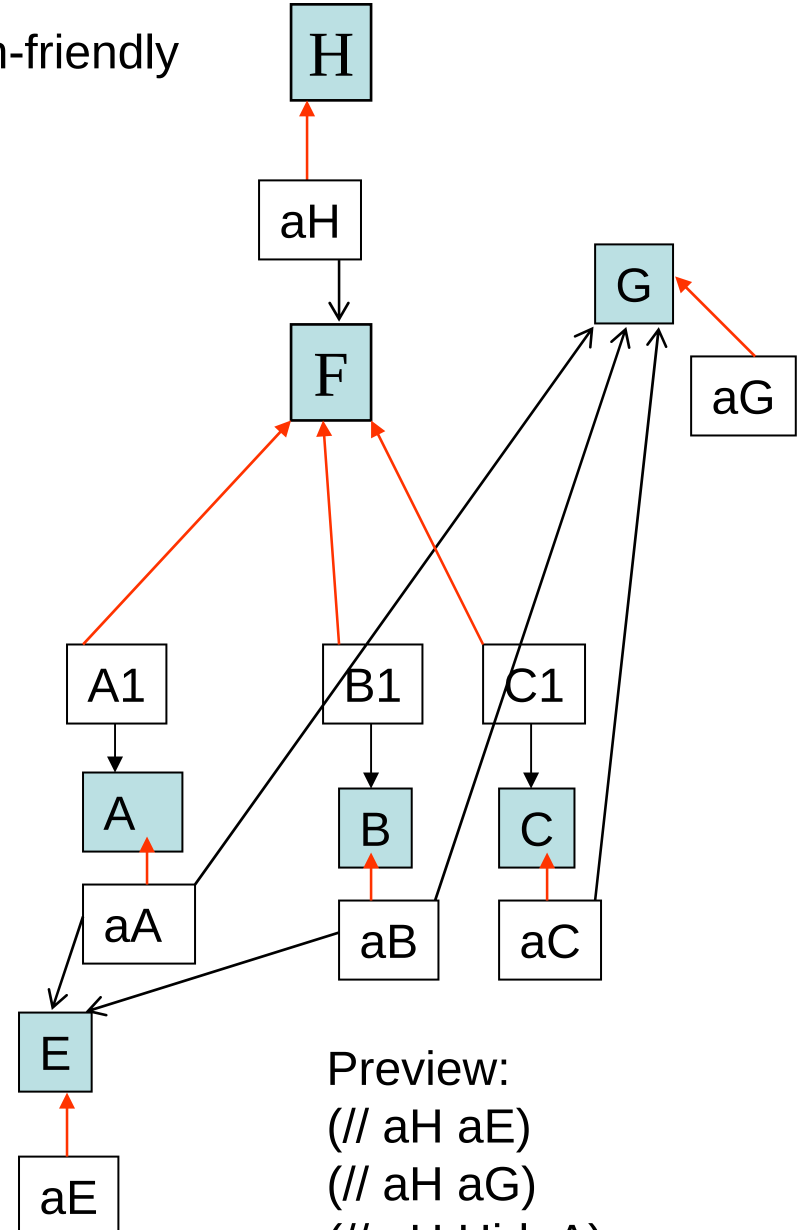

H
F
G
A
B
C
E
inheritance


H
aH
G
F
aG
Hid_A
Hid_B
Hid_C
B
E
aB
aE
not evolution-friendly

Class graphs

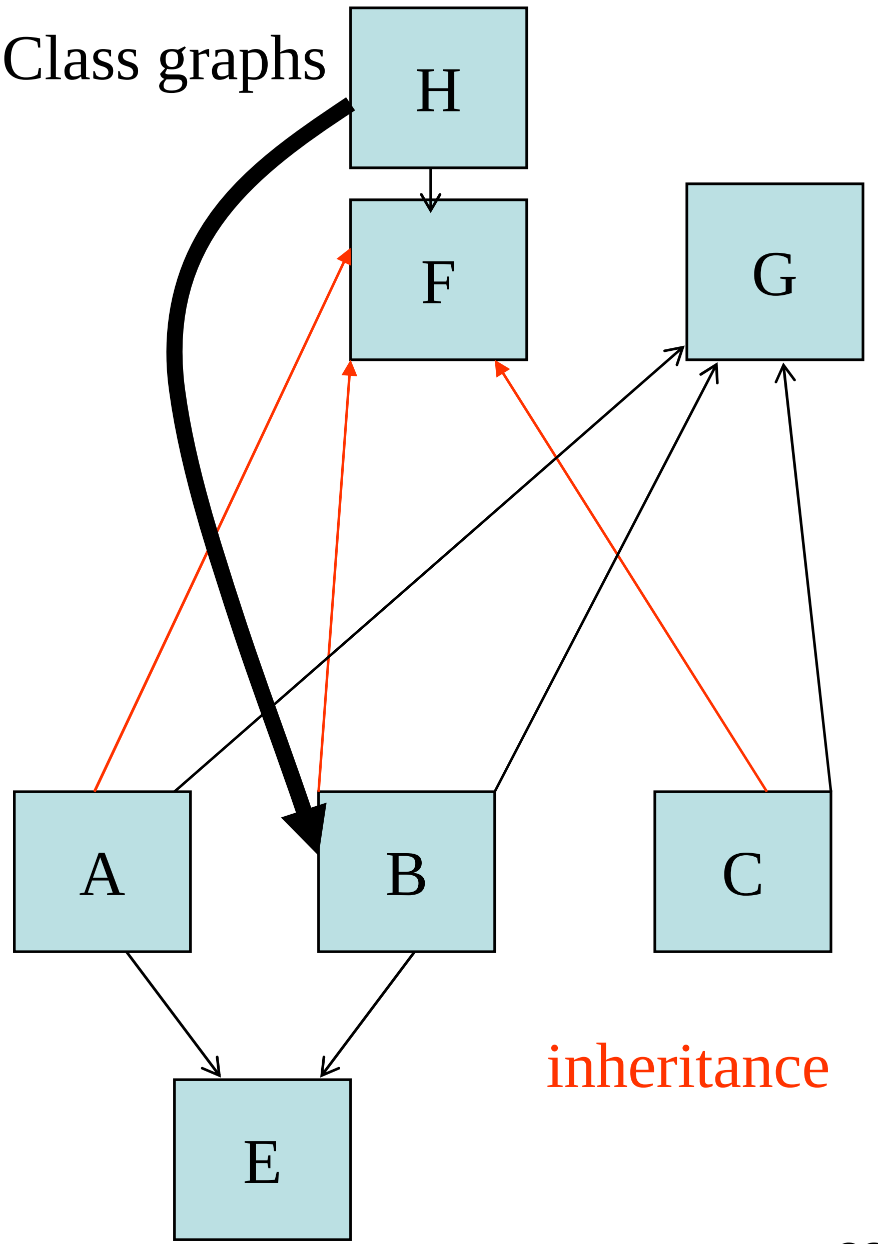


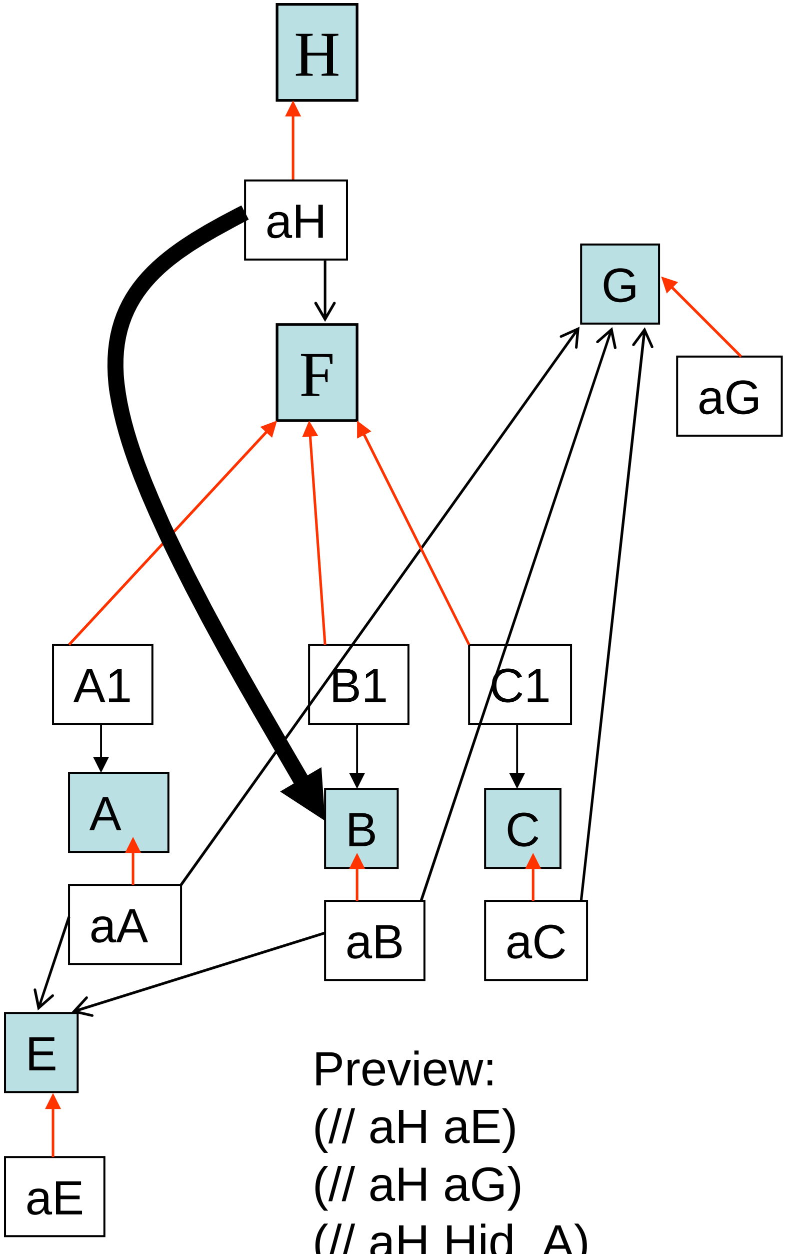
now evolution-friendly



Preview:
(// aH aE)
(// aH aG)
(// aH Hid_A)

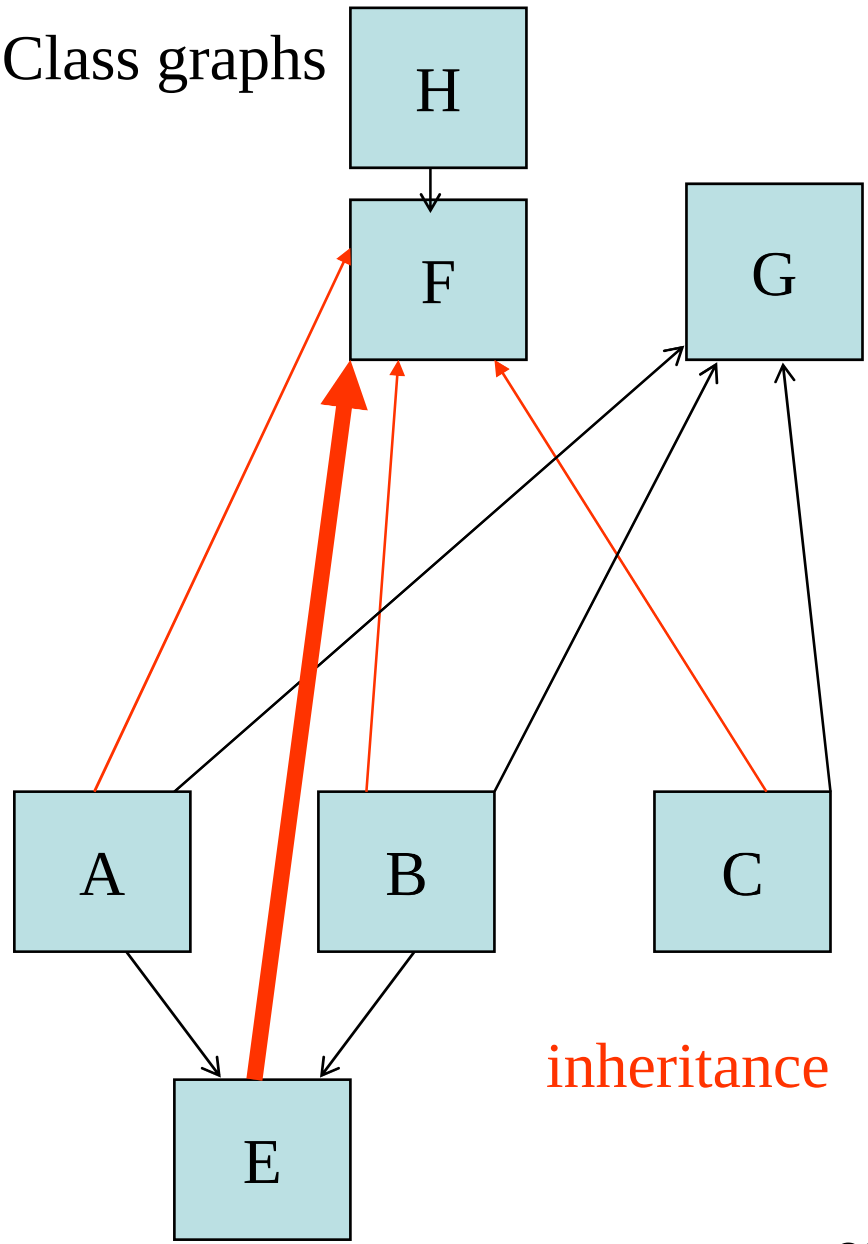
Class graphs

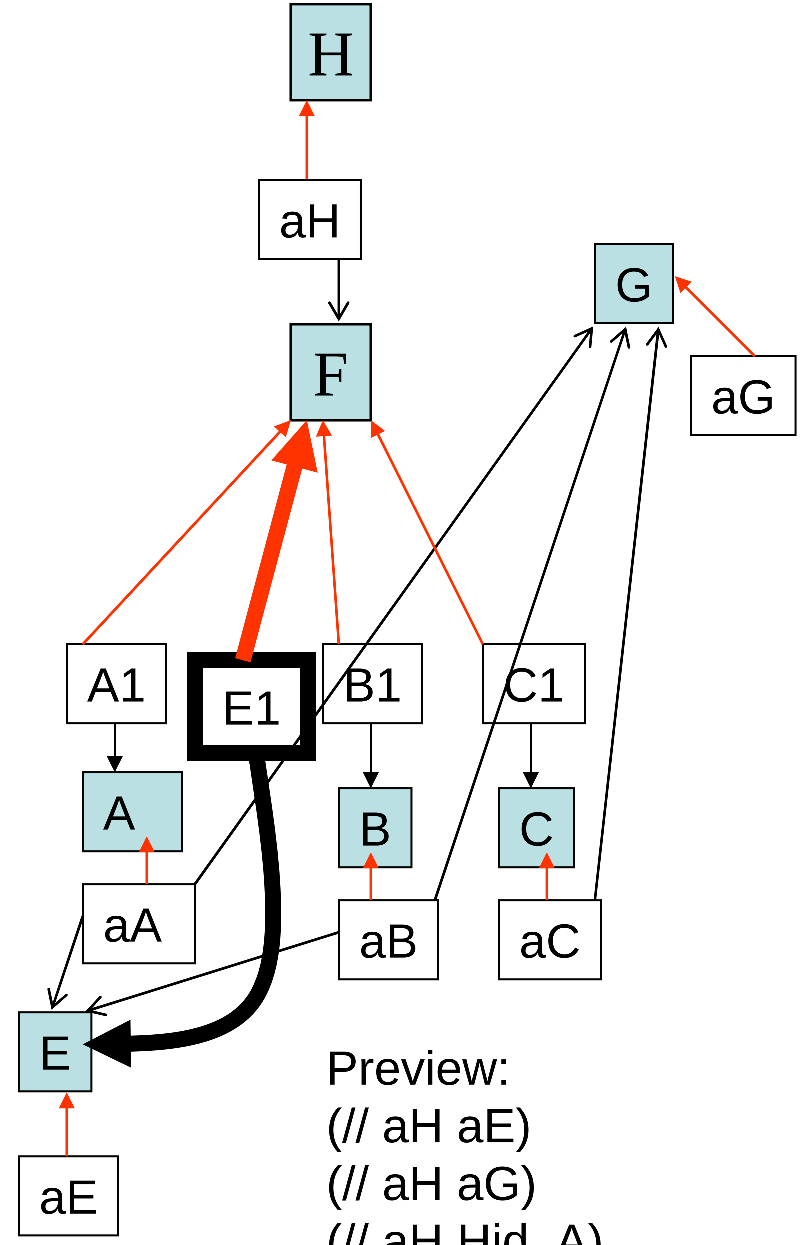




Preview:
(// aH aE)
(// aH aG)
(// aH Hid_A)

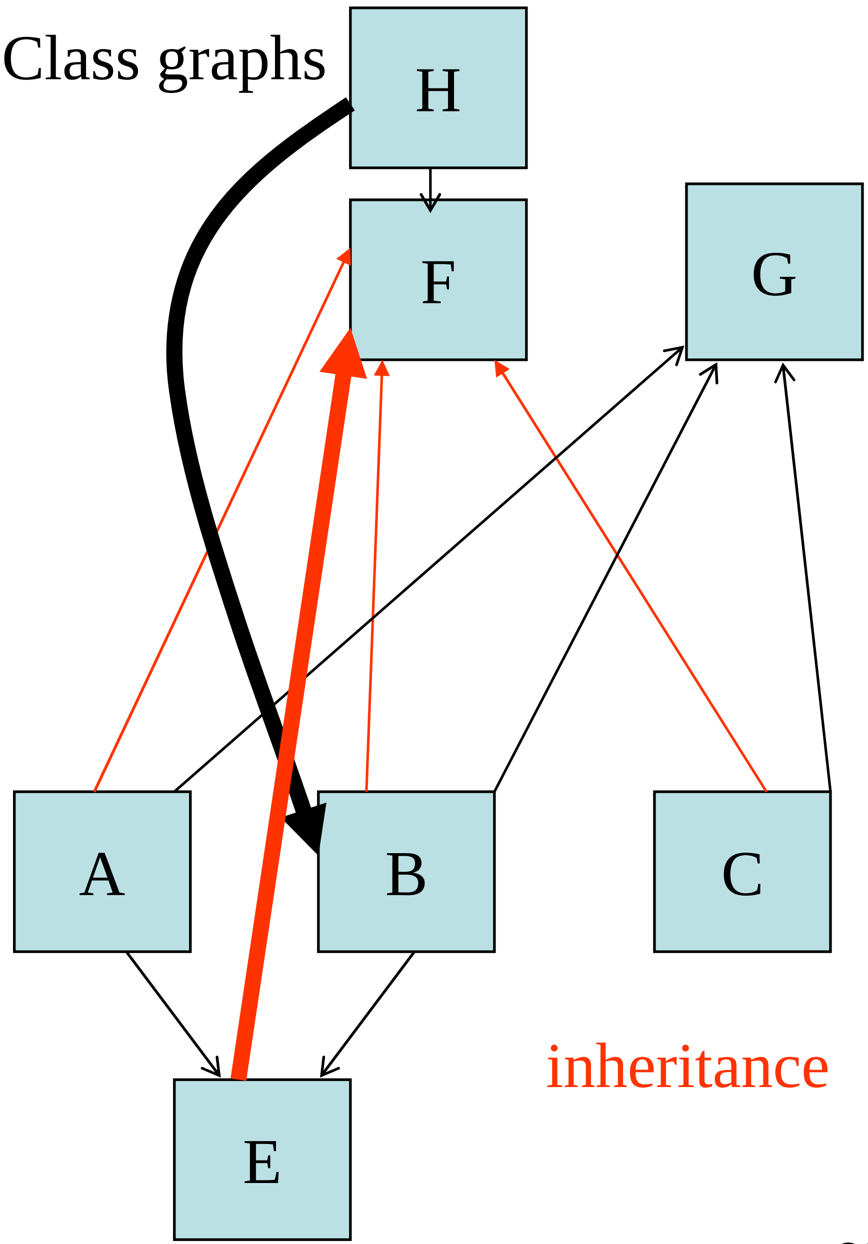
Class graphs





Preview:
(// aH aE)
(// aH aG)
(// aH Hid_A)

Class graphs



(datatype H (aH (f F) ***(b B)***))
(datatype G (aG))
(datatype A (aA (e E) (g G)))
(datatype B (aB (e E) (g G)))
(datatype C (aC (g G)))
(datatype E (aE))
(datatype F
  (A1 (a A))
  (B1 (b B))
  (C1 (c C))
  (E1 (e E)))

H = F ***B***.
G = .
A = E G.
B = E G.
C = G.
E = .
F : A | B | C ***| E*** .

Separate Viewgraphs

- Difference between homework class graphs and class graphs.
- No inheritance in homework class graphs.
- Flat class graphs can easily be modeled by home work class graph. A class graph is flat if abstract classes have no outgoing has-a edges. Quadratic growth in size.

Apply class graph knowledge to homework class graphs

- Only consider flat class graph (flattening is an object preserving transformation).
- In flat class graph the rules are simpler.

HW class graph to class graph transformation:
TypeName -- abstract class
AlternativeName – concrete class
(AlternativeName, FieldName, TypeName) – has-a
(AlternativeName, enclosing TypeName) – is-a

Meaning of strategies and visitors

- (// A B) (only this in hw)
  - A:AlternativeName B:AlternativeName
  - starts at A-object and ends at B-object
- (// A B)
  - A:TypeName B:TypeName
  - starts at an AlternativeName-object of A
  - ends at an AlternativeName-object of B

From Semantics to Interpreter

- From object-level semantics to class-level semantics
- (M1(M2 s cg) og)
  - M2: FIRST sets at class level

SWITCH to navig-object-graphs

From Interpreter to Compiler

- Connect to Structural Recursion
- Consider the strategy (// A *) (everything reachable from A)
- (M1(M2 s cg) og): we want M1 to be apply
- M2 must return a function that we apply to og
- Primitives: functions with one argument: the data traversed, no other arguments.

Code generation: should produce something useful

```
(define-datatype BusRoute BusRoute?
  (a-BusRoute
    (name symbol?)
    (buses (list-of Bus?))
    (towns (list-of Town?))))
```

Style 1: display

```
(define (trav br)
  (cases BusRoute br
    (a-BusRoute (name buses towns)
      (list name (trav-buses buses)
                  (trav-towns towns)))))
(define (trav-buses lob)
  (map trav-bus lob))
```

Style 2: copy

- (define (cp br)
- (cases BusRoute br
- (a-BusRoute (name buses towns)
  (apply a-BusRoute (list name (cp-buses
  buses) (cp-towns towns))))))
- (define (cp-buses lob)
- (map cp-bus lob))

Summary phase 1

- Language: strategies: A // B, class graphs, object graphs
- Semantics: FIRST: there exists object
- Interpreter: FIRST: there exists path in class graph
- Compiler: generated code is equivalent to a subgraph of class graph

Visitors

Visitors

- Several kinds:
  - Think of strategy as making a list out of an og. Fold on that list.
    - (cg og s) -> list of target objects of s. (gather cg og s). (// CContainer CWeight)
    - (+ (+   …   (+ w2 (+ w1 0)) …)
  - Think of visitor as having a suit case of variables in which they store data from their trip. Available as argument.
  - functions for nodes and edges.
  - multiple visitors.

Type checking of hw programs

- check: Program =(Strategy x Visitor). (Program x ClassGraph) -> Bool
- Fundamental question: Given a program, with respect to which class graphs is it type correct.
  - Type checking: Given a class graph, is the program type correct?
  - Typability: Does there exist a class graph such that the program is type correct?

Reference

- Class-graph Inference for Adaptive Programs, Jdens Palsberg, TAPOS 3 (2), 75-85, 1997.