From: “Это все придумал Черчиль в 18 году” (C) Высоцкий В.С
To: “Это все придумал Мейер в 85 году” ☺

Incidence matrix and OOP

Alexey Kanatov,
alexey.v.kanatov@gmail.com
LinkedIn

Content

- Brief personal introduction and motivation of the work
- Basic terms and foundations
- General algorithm
- Outcome
- Dynamic loading of objects of statically unknown classes
- Summary

<u>Disclaimer:</u> not all topics are fully investigated and some are partially covered. Separate talks may be provided to cover

Personal introduction

- 10+ years in compilers (Modula-2, Ada, Eiffel, Accord, STS)
- 15+ years SW R&D and general management (Intel, Samsung, WorldQuant)
- 4 years teaching at MEPhI, school #548, Innopolis University
- My advisors, role models
  - Стрижевский В.С. – Модула-2
  - Перминов О.Н. – Ada
  - Meyer B – Eiffel
- "My way"
  - Huawei, Chief academic consultant ☺
  - Innopolis University, Associate pprofessor, lab head
  - Samsung, Compiler, Platform, System AI Tools department head
  - WorldQuant Research (Eurasia), director
  - Intel, head of Compiler QA, Compiler Russia, Moscow Site, Intel Platform Simulator
  - Object Tools Inc., Visual Eiffel compiler architect and key developer

Motivation and objective

- 1993-96 – do not do VMT, do 'FST' I was told – was it a right command? <u>Doubt</u>

- 1993-96 – I draw a matrix with classes vs. origin&seed – worth to deepen analysis of the topic? <u>Not all was done 30 years ago</u>

- Inheritance is bad, dynamic dispatch is heavy, fragile base class – a lot of **educated** believes. <u>А баба Яга против</u>☺

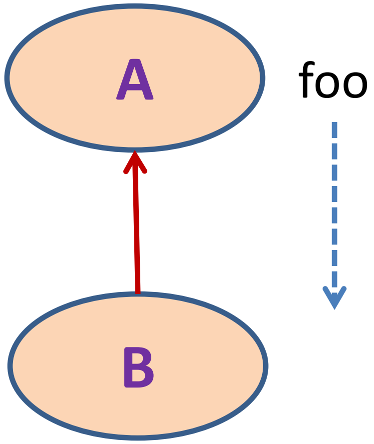
- What I remember from discreet math course – matrix rows and columns can be swapped ☺ <u>Your feedback is welcome!</u>

Basic terms

- Object is a set of attributes. Objects with identical set of attributes' kinds form a type which is described by class
- Class is … a named collection of members (features, characteristics)
  - Member can be routine (function) or attribute (field)
    - Routine can be procedure (action, command) or function (query)
    - Attribute (query) can be variable or constant
  - Another view: there are only attributes – variable or constant (assigned once). Actions (routines) are just constant attributes of the function type
- Origin is the class the member was initially declared
- Seed is the initial member declaration in the origin class
- Inheritance – relation between classes implying all members of every parent 'go down' to the child class. Base-derived, supertype, extension – no need to step into terminology discussion
- Version of the member – in some class we may have several versions – coming from the same origin&seed under the same or different names, form different ones under the same name

Foundations (I): inheritance basics

1.

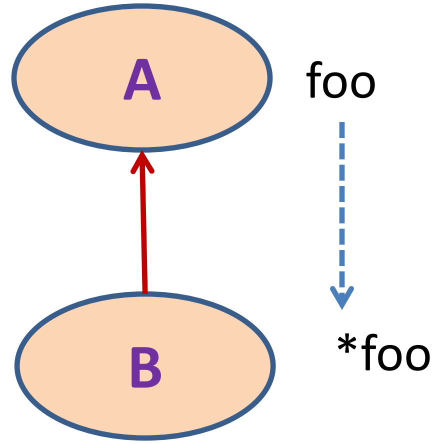



```
class A
   foo
end
class B inherit A
end
```

foo$A

Class	Version
A	foo@A
B	foo@A

2.



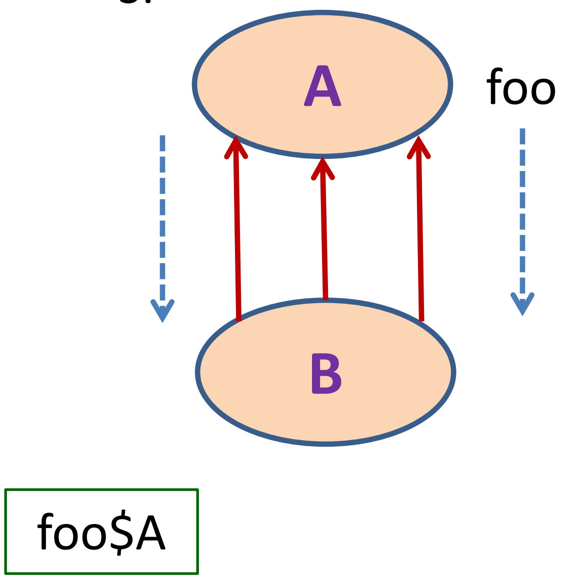

```
class A
   foo
end
class B inherit A
   override foo
end
```

foo$A

Class	Version
A	foo@A
B	foo@B

Foundations (II): no replication, but merge

3.

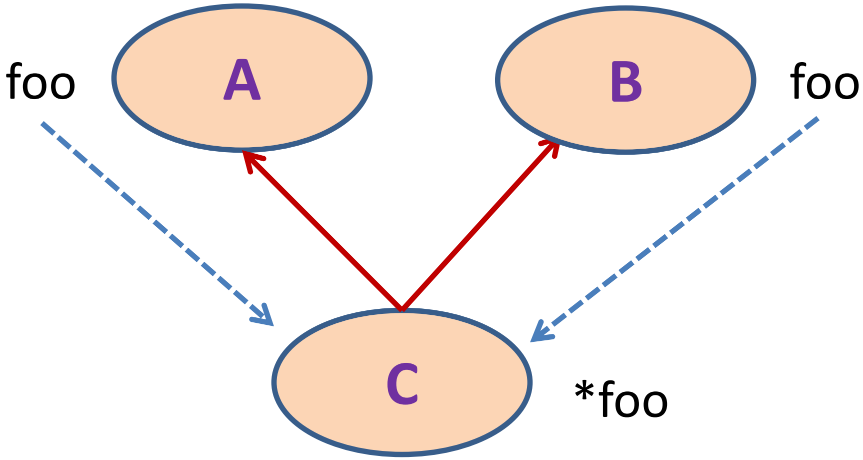


Class	Version
A	foo@A
X	foo@A
B	foo@A

```
/* There could be many
paths from B to A, with
many classes on all these
paths */

class A
   foo
end
class X inherit A
end
class B inherit A, A, X
end
```

Foundations (III): kill many birds with one stone

4.

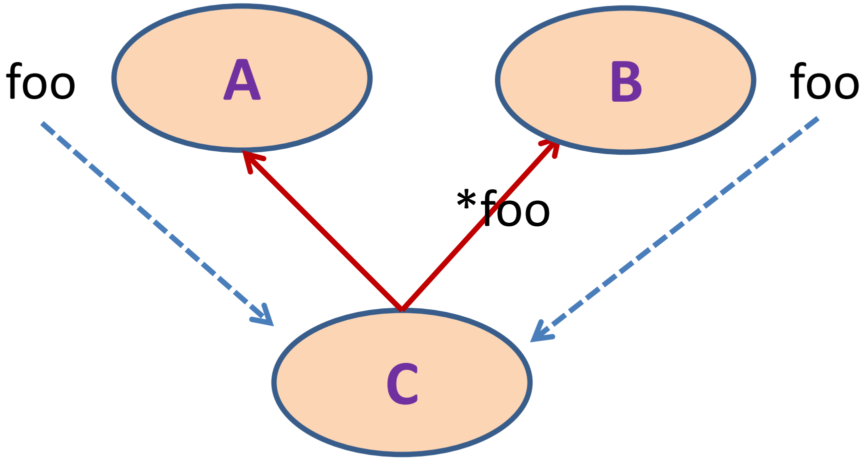


Class	foo$A	foo$B
A	foo@A	
B		foo@B
C	foo@C	foo@C

```
class A
    foo
end
class B
    foo
end
class C inherit A, B
    override foo
end
```

Foundations (IV): kill many birds with one stone

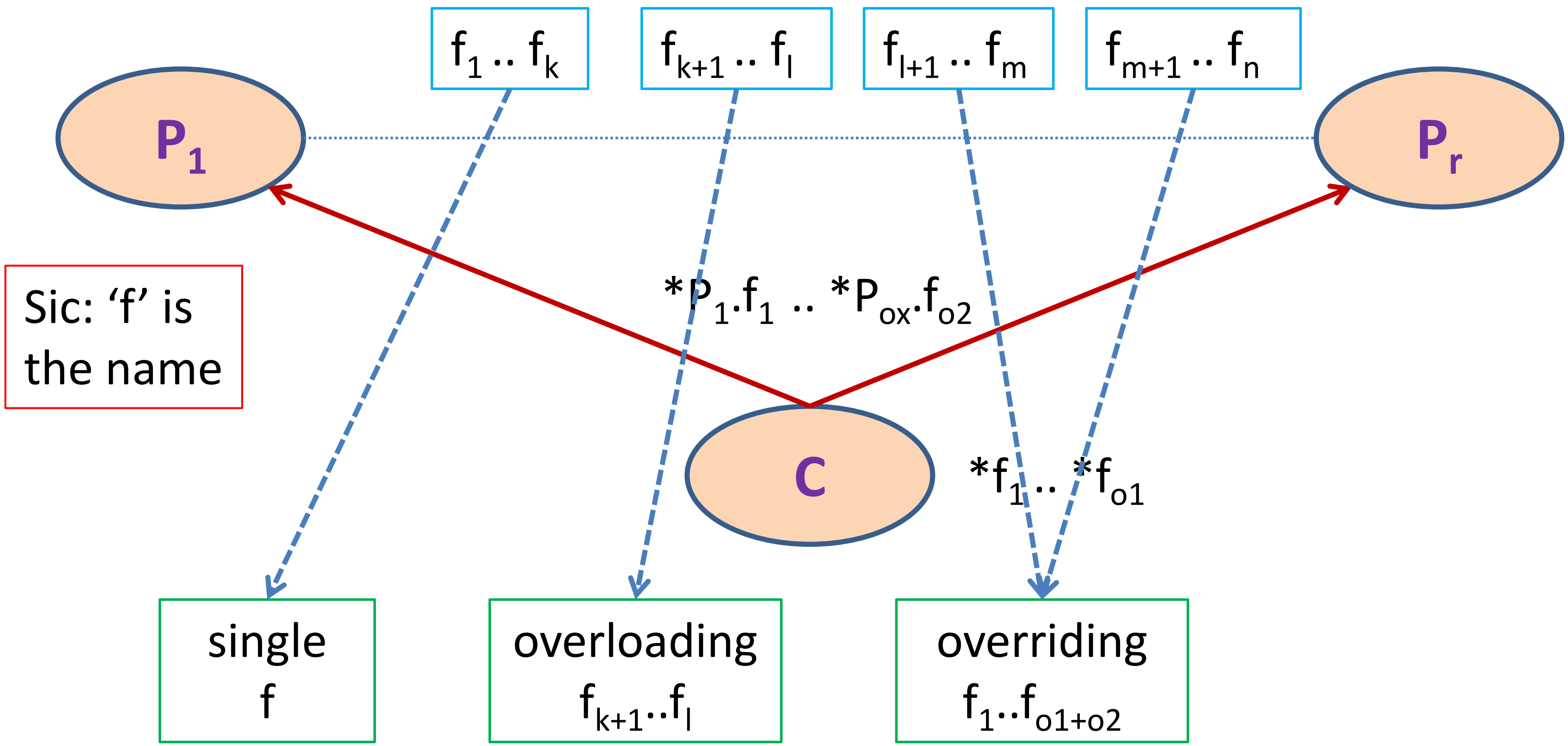
5.



Class	foo$A	foo$B
A	foo@A	
B		foo@B
C	foo@B	foo@B

```
class A
    foo
end
class B
    foo
end
class C inherit A, B
    override B.foo
end
```

Foundations (V): generalization, no replication, kill many birds with many stones



Foundations (VI): any graph can be presented as the incidence matrix

this ->

	s\$O$_1$	s\$O$_2$	s\$O$_3$	…	s\$O$_m$
C_1	v@C_{11}				
C_2		v@C_{22}			
C_3	v@C_{13}				
…					
C_n	v@C_{1n}		v@C_{3n}		

- matrix is sparse!
- matrix contains addresses for routines and offsets from **this** for fields
- inheritance graph has the sink – Any (Object)
- treat this matrix as rows – VMT-like approach, vector indexed by origin$seed ID (1 .. m) –> direct access to EA (effective address)
- treat this matrix as columns – MST approach, vector indexed by object class ID (1 .. n) –> direct access to EA

Foundations (VII): any member activation will look like

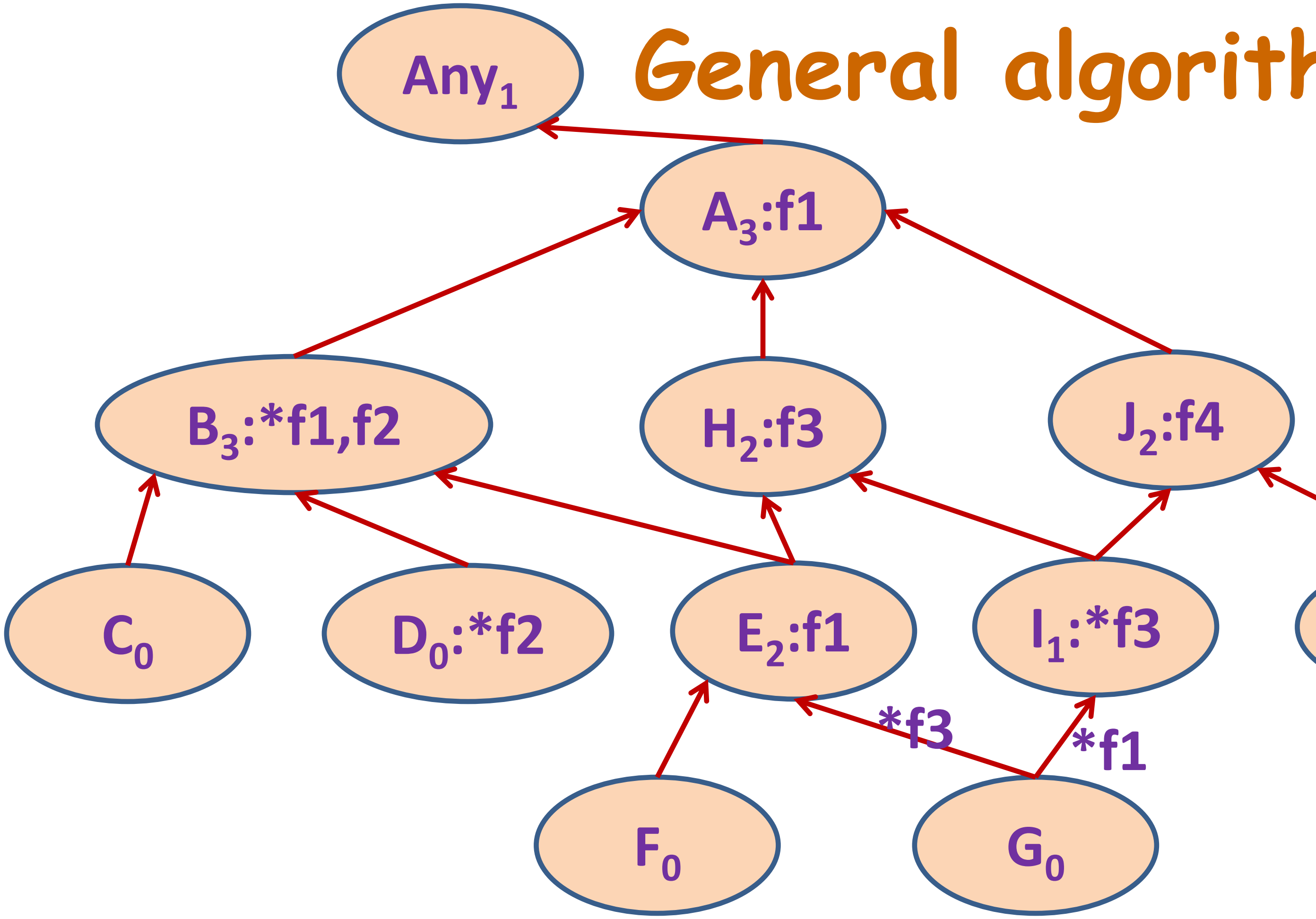
```
// Source code
target1.foo ()
target2.field1 := target3.field2

// Pseudo-asm code: row view
call target1[foo:seed$origin]
load target3[field2:seed$origin], #R1
store #R1, target2[field1:seed$origin]

// Pseudo-asm code: column view
call foo:seed$origin [target1]
load field2:seed$origin [target3], #R1
store #R1, field1:seed$origin [target2]
```

- there will be difference in number of instructions and their nature for row and column based approaches for real assemblers! Rows are better
- matrix is sparse – how to keep direct access and get rid of empty cells

Foundations (VIII): can we optimize the matrix?

- Remove rows – no objects of the class at runtime
  - Abstract classes
  - Class does not belong to dynamic class sets (needs full program analysis)
- Empty cells – particular version is never activated (fields caveat)
  - Dead-code elimination in case of OOP (needs full program analysis)
- Remove columns
  - The same non-empty value in the column
- Assume we did all that –> what's next –> to reorganize the matrix

General algorithm: demo

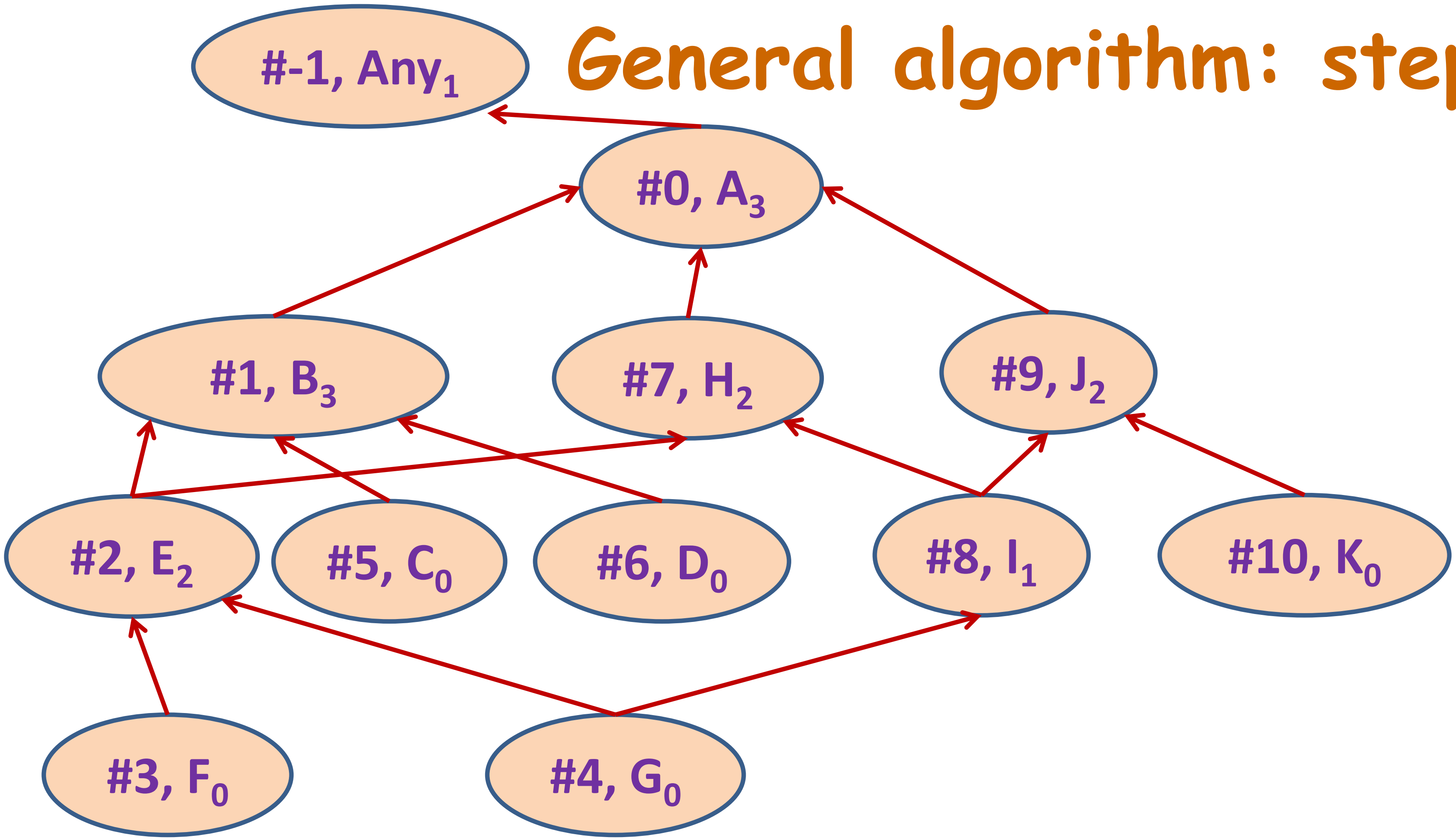


* - stands for **override** in class or while inheriting

X_n – means number of children the class has

Sort by number of children at every level

General algorithm: steps



- Numerate classes starting from 0
- Abstract or 'objectless' class will get -1

General algorithm: columns outcome

	$f_1\$A$	$f_2\$B$	$f_3\$H$	$f_4\$J$
#0, A	$f_1@A$			
#1, B	$f_1@B$	$f_2@B$		
#2, E	$f_1@E$	$f_2@B$	$f_3@H$	
#3, F	$f_1@E$	$f_2@B$		
#4, G	$f_1@A$	$f_2@B$	$f_3@H$	$f_4@J$
#5, C	$f_1@B$	$f_2@B$		
#6, D	$f_1@B$	$f_2@D$		
#7, H	$f_1@A$		$f_3@H$	
#8, I	$f_1@A$		$f_3@I$	$f_4@J$
#9, J	$f_1@A$			$f_4@J$
#10, K	$f_1@A$			$f_4@K$

Columns' view: <u>no empty cells</u>, <u>no direct access</u>

<u>$f_1\$A$</u>:

A, G, H, I => $f_1@A$,
B, C, D => $f_1@B$,
E, F => $f_1@E$

<u>$f_2\$B$</u>:

B, E, F, G,C=> $f_2@B$,
D => $f_1@D$

<u>$f_3\$H$</u>:

E, G, H => $f_3@H$,
I => $f_3@I$

<u>$f_4\$J$</u>:

G, I, K => $f_4@J$,
K => $f_4@K$

General algorithm: columns outcome

	$f_1\$A$	$f_2\$B$	$f_3\$H$	$f_4\$J$
#0, A	f_1@A			
#1, B	f_1@B	f_2@B		
#2, E	f_1@E	f_2@B	f_3@H	
#3, F	f_1@E	f_2@B		
#4, G	f_1@A	f_2@B	f_3@H	f_4@J
#5, C	f_1@B	f_2@B		
#6, D	f_1@B	f_2@D		
#7, H	f_1@A	**1**	f_3@H	
#8, I	f_1@A		f_3@I	f_4@J
#9, J	f_1@A		**2**	f_4@J
#10, K	f_1@A			f_4@K
	0			**4**

EA = **this** -> class ID + MST -> shift

Direct access + some address arithmetic burden

General algorithm: rows outcome

	$f_1\$A$	$f_2\$B$	$f_3\$H$	$f_4\$J$
#0, A	$f_1@A$			
#1, B	$f_1@B$	$f_2@B$		
#2, E	$f_1@E$	$f_2@B$	$f_3@H$	
#3, F	$f_1@E$	$f_2@B$		
#4, G	$f_1@A$	$f_2@B$	$f_3@H$	$f_4@J$
#5, C	$f_1@B$	$f_2@B$		
#6, D	$f_1@B$	$f_2@D$		
#7, H	$f_1@A$		$f_3@H$	
#8, I	$f_1@A$		$f_3@I$	$f_4@J$
#9, J	$f_1@A$			$f_4@J$
#10, K	$f_1@A$			$f_4@K$

Rows' view: empty cells, direct access

'Smart' rows' view - 2 kinds of vectors:

- Fast – fully filled, direct access
- Compact – no empty cells, no direct access

H:

$f_1\$A \Rightarrow f_1@A$,
$f_3\$H \Rightarrow f_3@H$

Delta to switch from Fast to Compact

Indication of potential dynamic class loading case

- Pattern of class loading

```
foo (<parameters>): ReturnType foreign
```

- What to be stored in meta and what to be rebuilt?

	$f_1\$A$	$f_2\$B$	$f_3\$H$	$f_4\$J$
#0, A	f_1@A			
#1, B	f_1@B	f_2@B		
#2, E	f_1@E	f_2@B	f_3@H	
#3, F	f_1@E	f_2@B		
#4, G	f_1@A	f_2@B	f_3@H	f_4@J
#5, C	f_1@B	f_2@B		
#6, D	f_1@B	f_2@D		
#7, H	f_1@A		f_3@H	
#8, I	f_1@A		f_3@I	f_4@J
#9, J	f_1@A			f_4@J
#10, K	f_1@A			f_4@K

One new class:

- One new row
- Potentially several new columns

Aim: no difference between access to objects of classes known at compile time and ones loaded dynamically

Summary

Incidence matrix class vs. seed&origin represents well the whole inheritance graph. It is the central data structure for analysis and optimizations

Classes numbering scheme based on the nature of the inheritance graph and seed&origin numbering scheme based on the length of the column vectors delivers blocked matrix which supports direct access with minimal memory losses to store empty cells

Dynamic loading of new classes enforces keeping meta information to rebuild the matrix and regenerate a lot of code in the worst case

Thank you !

Q&A