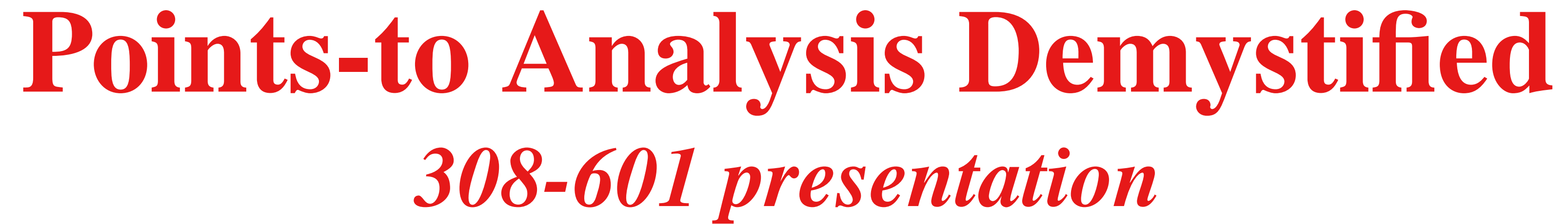


Ondřej Lhoták

April 24, 2002

Introduction

If points-to analysis doesn't confuse you...

Introduction

If points-to analysis doesn't confuse you...

you just haven't seen enough of it yet!

$$\frac{\mathbf{x} : \mathbf{ref}(_ \to \alpha) \quad \mathbf{y} : \mathbf{ref}(_ \to \alpha)}{\mathit{welltyped}(\mathbf{x} = \mathbf{y})}$$

$$\frac{\mathbf{x} : \mathbf{ref}(_ \to (\tau \times _)) \quad \mathbf{y} : \tau}{\mathit{welltyped}(\mathbf{x} = \&\mathbf{y})}$$

$$\frac{\mathbf{x} : \mathbf{ref}(_ \to \alpha) \quad \mathbf{y} : \mathbf{ref}(_ \to (\mathbf{ref}(_ \to \alpha) \times _))}{\mathit{welltyped}(\mathbf{x} = *\mathbf{y})}$$

$$\frac{\begin{array}{c}\mathbf{x} : \mathbf{ref}(_ \to \alpha) \\ \mathbf{y}_1 : \mathbf{ref}(_ \to \alpha) \ \ldots \ \mathbf{y}_n : \mathbf{ref}(_ \to \alpha)\end{array}}{\mathit{welltyped}(\mathbf{x} = \mathbf{op}(\mathbf{y}_1 \ \ldots \ \mathbf{y}_n))}$$

$$\frac{\mathbf{x} : \mathbf{ref}(_ \to (\mathbf{ref}(_ \to \alpha) \times _)) \quad \mathbf{y} : \tau}{\mathit{welltyped}(\mathbf{x} = \mathbf{allocate}(\mathbf{y}))}$$

$$\frac{\mathbf{x} : \mathbf{ref}(_ \to (\mathbf{ref}(_ \to \alpha) \times _)) \quad \mathbf{y} : \mathbf{ref}(_ \to \alpha)}{\mathit{welltyped}(*\mathbf{x} = \mathbf{y})}$$

Presentation Goals

- to help you (and me) understand points-to analysis in general

Presentation Goals

- to help you (and me) understand points-to analysis in general
- to compare some recent points-to analyses to the standard ones

References

- Das. Unification-based Pointer Analysis with Directional Assignments. PLDI 00.
- Liang, Pennings, Harrold. Extending and Evaluating Flow-insensitive and Context-insensitive Points-to Analyses for Java. PASTE 01.
- Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994.
- Steensgaard. Points-to Analysis in Almost Linear Time. POPL 96.

Outline

- Models of points-to analyses
- Flow-graph model
- Andersen's and Steensgaard's analyses
- One-level flow analysis [Das]
- Various analyses for Java [Liang et al]
- Conclusions

Analysis Goal

- Our focus is limited to analyses that are
  - May-point-to analyses
  - Flow-insensitive
  - Context-insensitive
- Goal: For each variable, determine allocation sites from which objects can reach it.

Models

- Points-to graph

Models

- Points-to graph
- Type inference

Models

- Points-to graph
- Type inference
- Algorithm

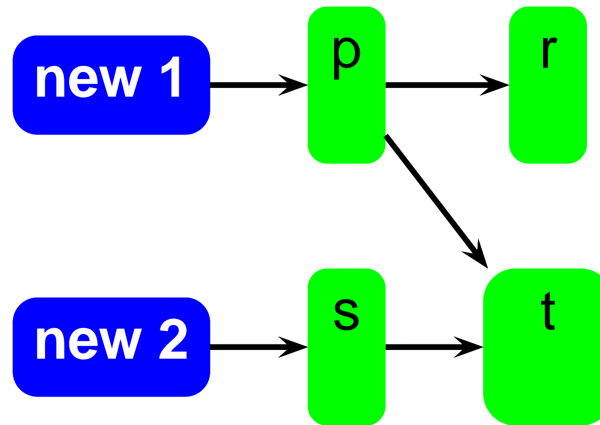
Models

- Points-to graph
- Type inference
- Algorithm
- Constraint graph

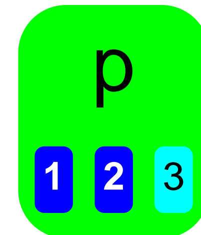
Models

- Points-to graph
- Type inference
- Algorithm
- Constraint graph
- Flow graph

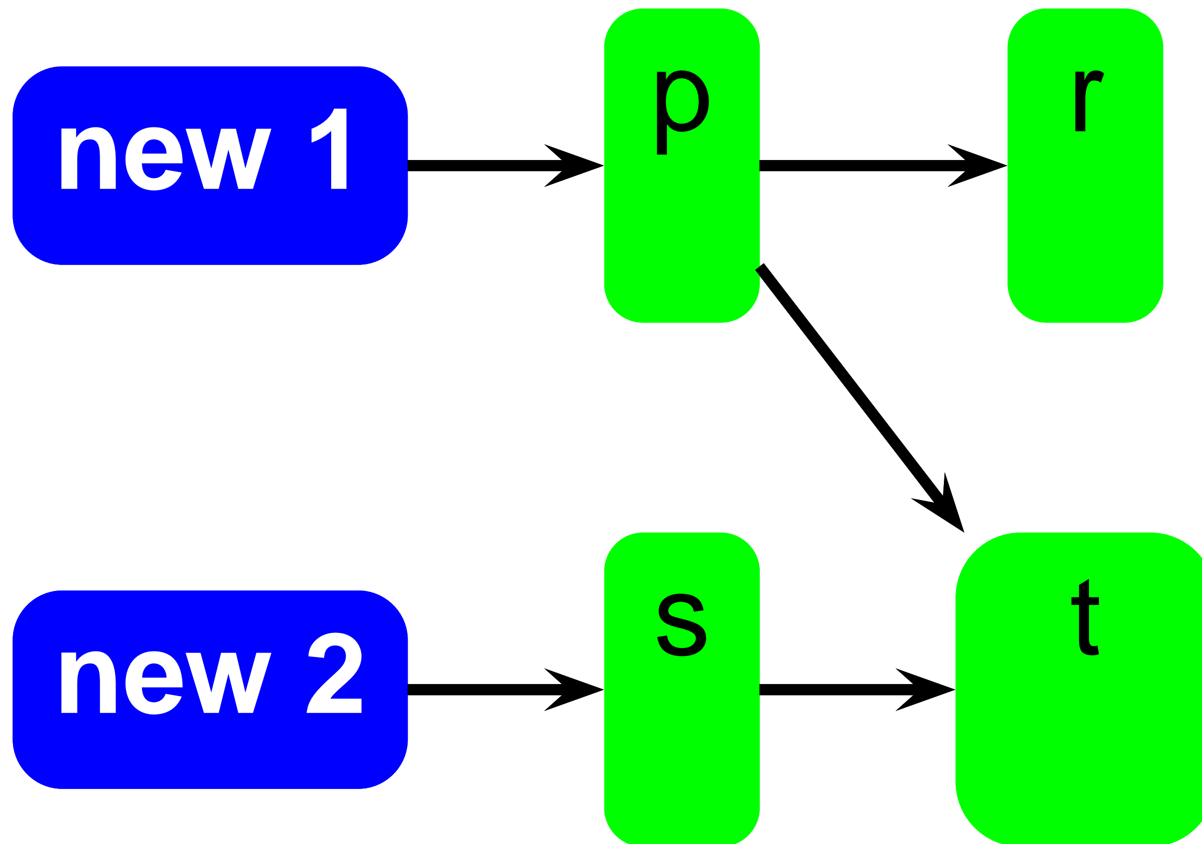
Flow graph



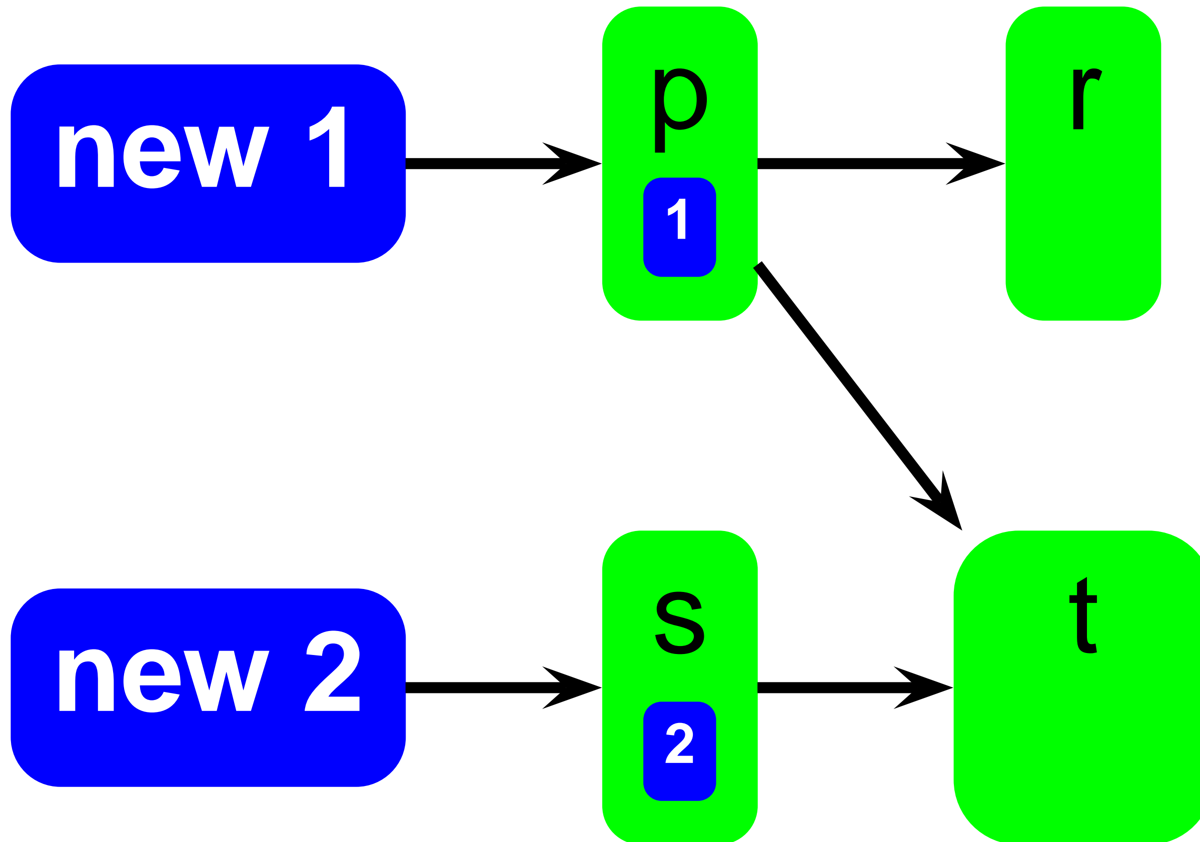
- Nodes represent references and locations
- Edges represent data flow (assignments)
- Nodes contain sets of reaching refs



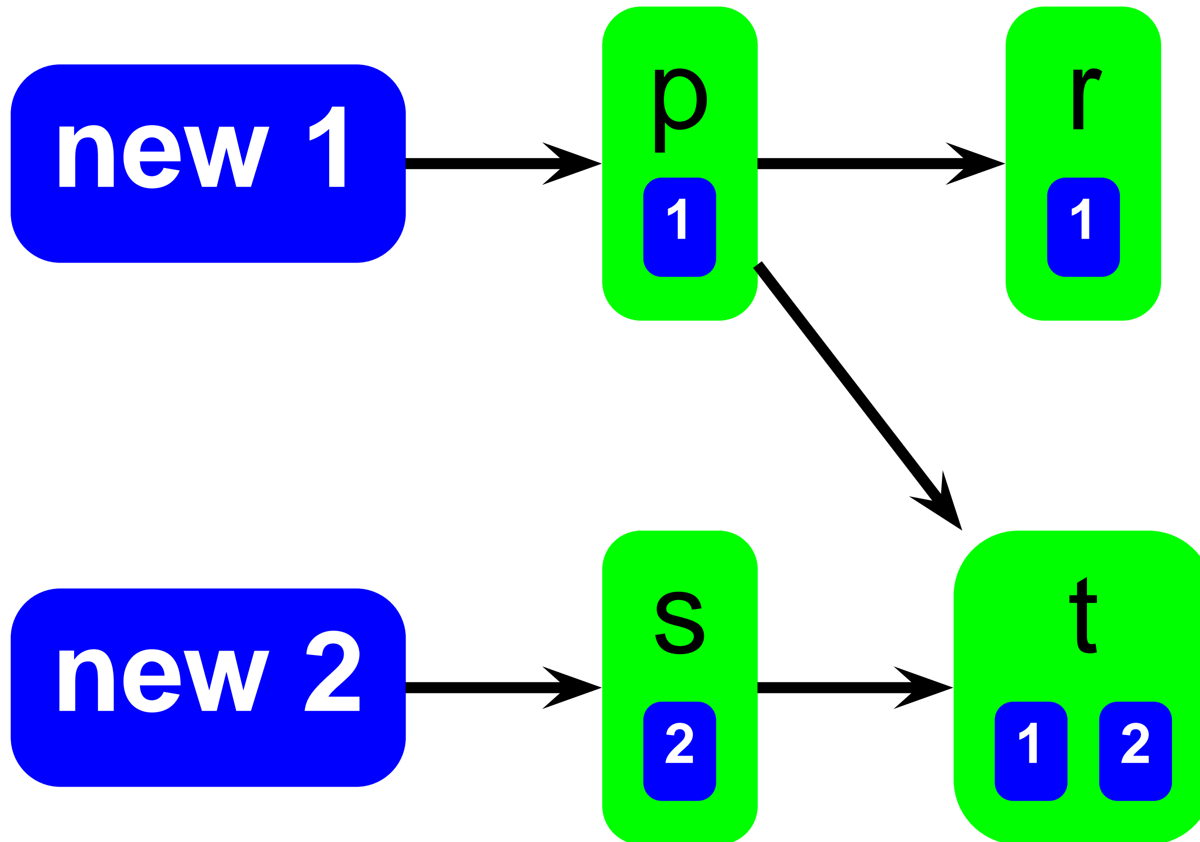
Flow graph



Flow graph



Flow graph



Node types

p

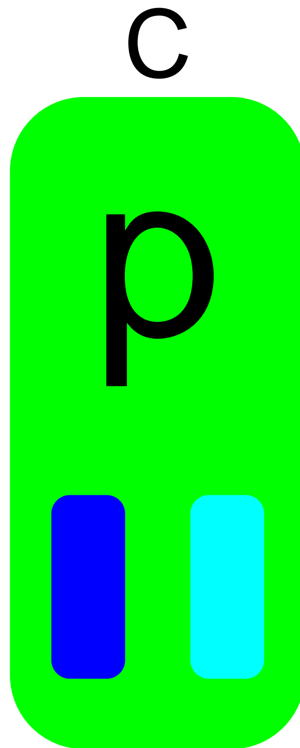
&p

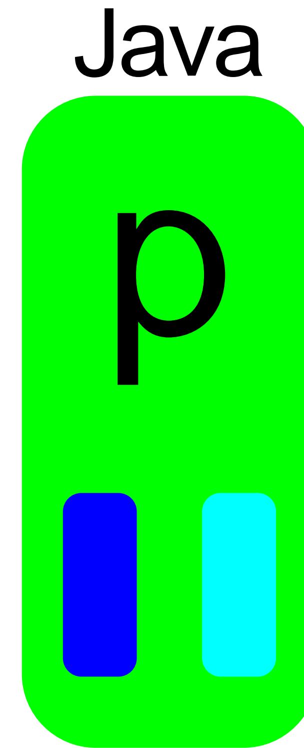
*p

new 1

*1

Node types





- Green nodes represent simple variables

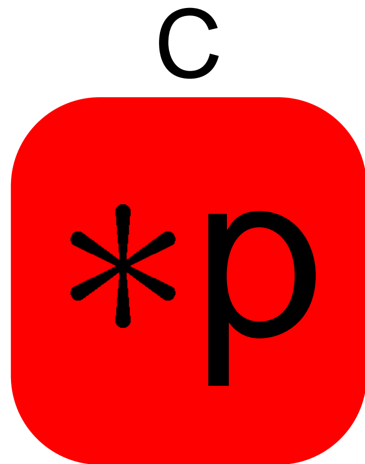
Node types

C

Java

- Cyan nodes represent addresses of variables

Node types



- Red nodes represent pointer dereferences

Node types

C

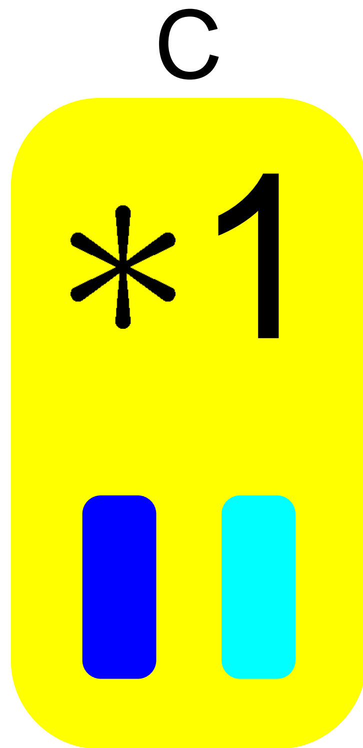
malloc 1

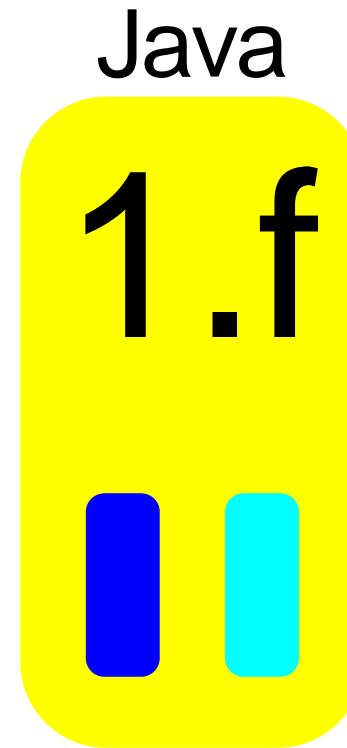
Java

new 1

- Blue nodes represent addresses of heap objects

Node types





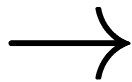
- Yellow nodes represent contents of heap objects

Normalizing programs

```
p = foo( q );


Object
foo( Object r )
{
    return r;
}
```

Normalizing programs

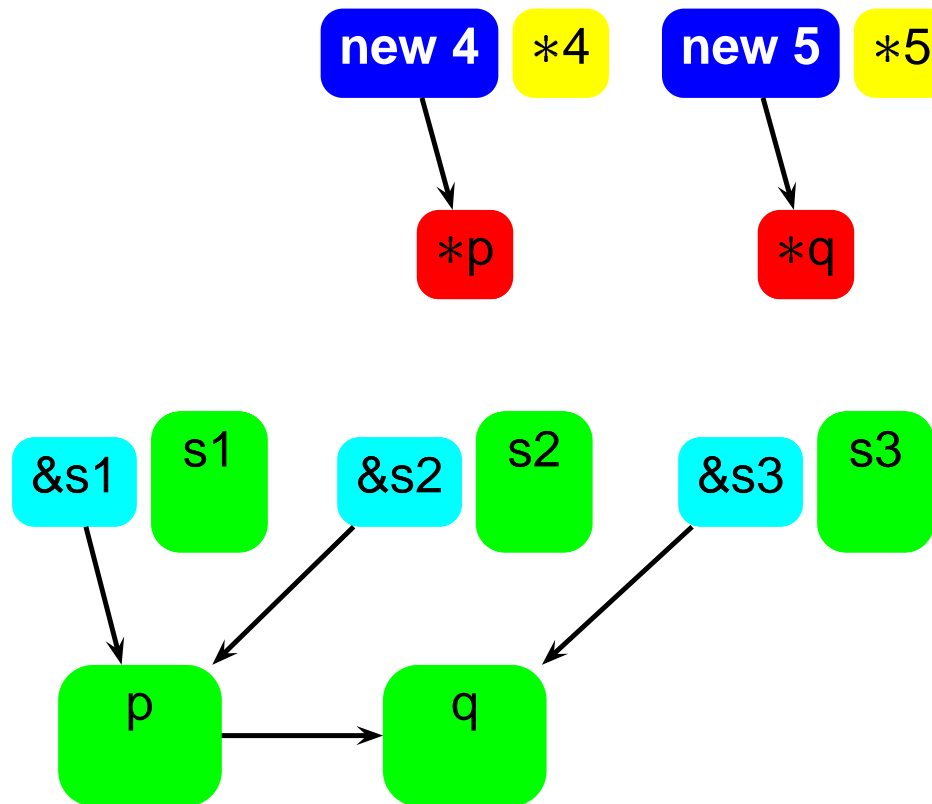
```
p = foo( q );

Object
foo( Object r )
{
    return r;
}
```



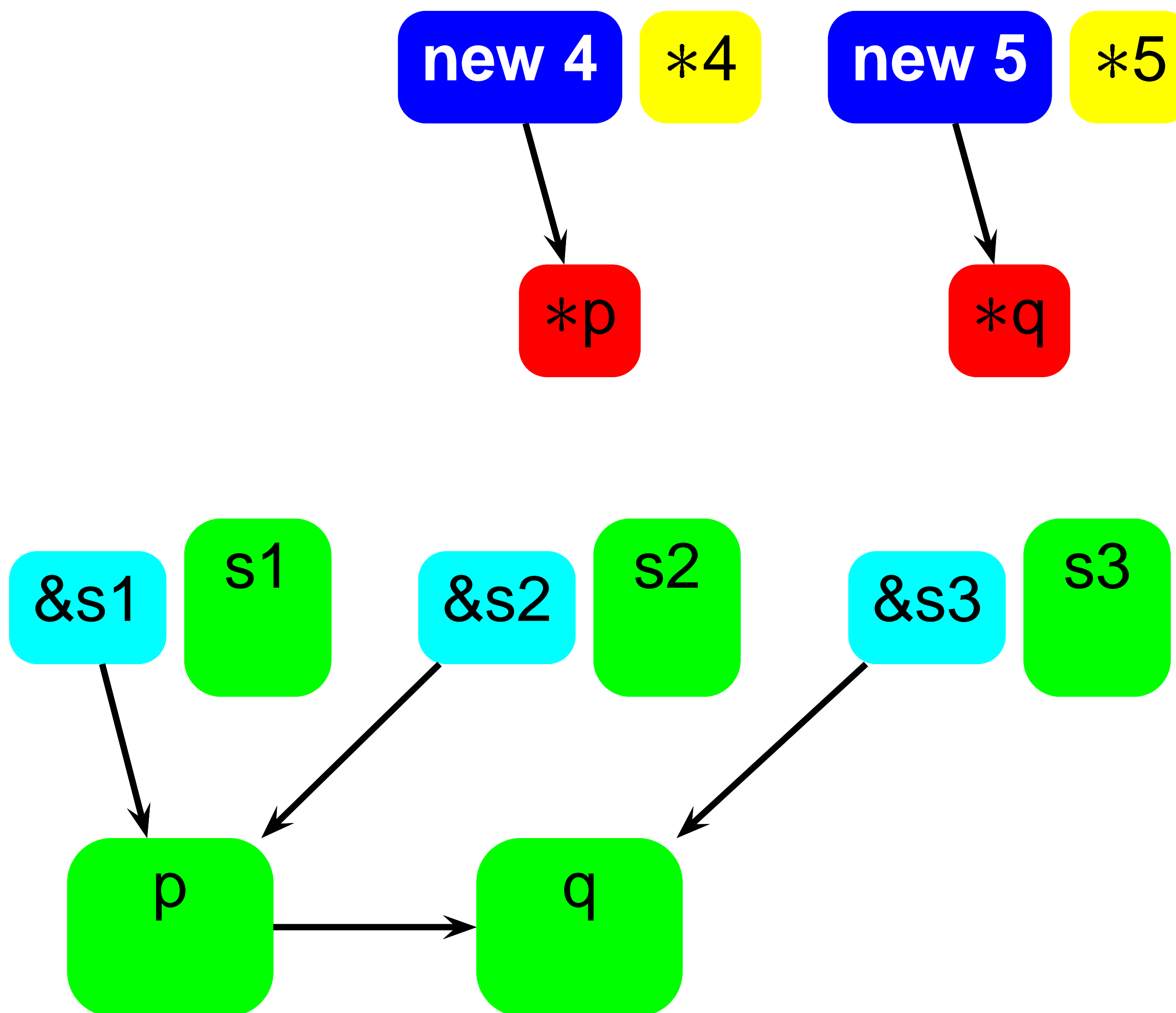
```
foo@1 = q;
r = foo@1;
foo@ret = r;
p = foo@ret;
```

Example

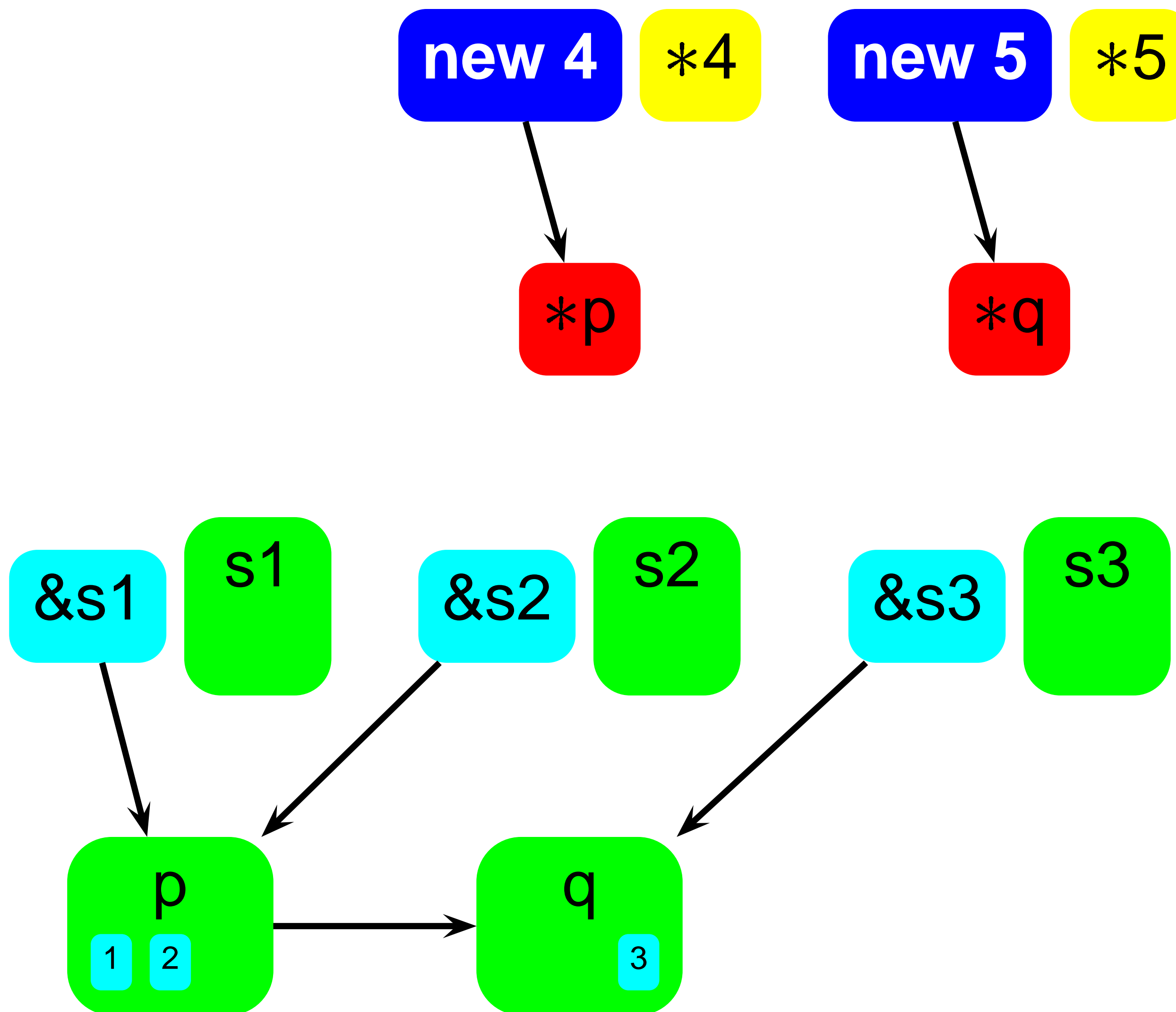
```
p = &s1;
p = &s2;
q = &s3;
q = p;
*p = new 4;
*q = new 5;
```



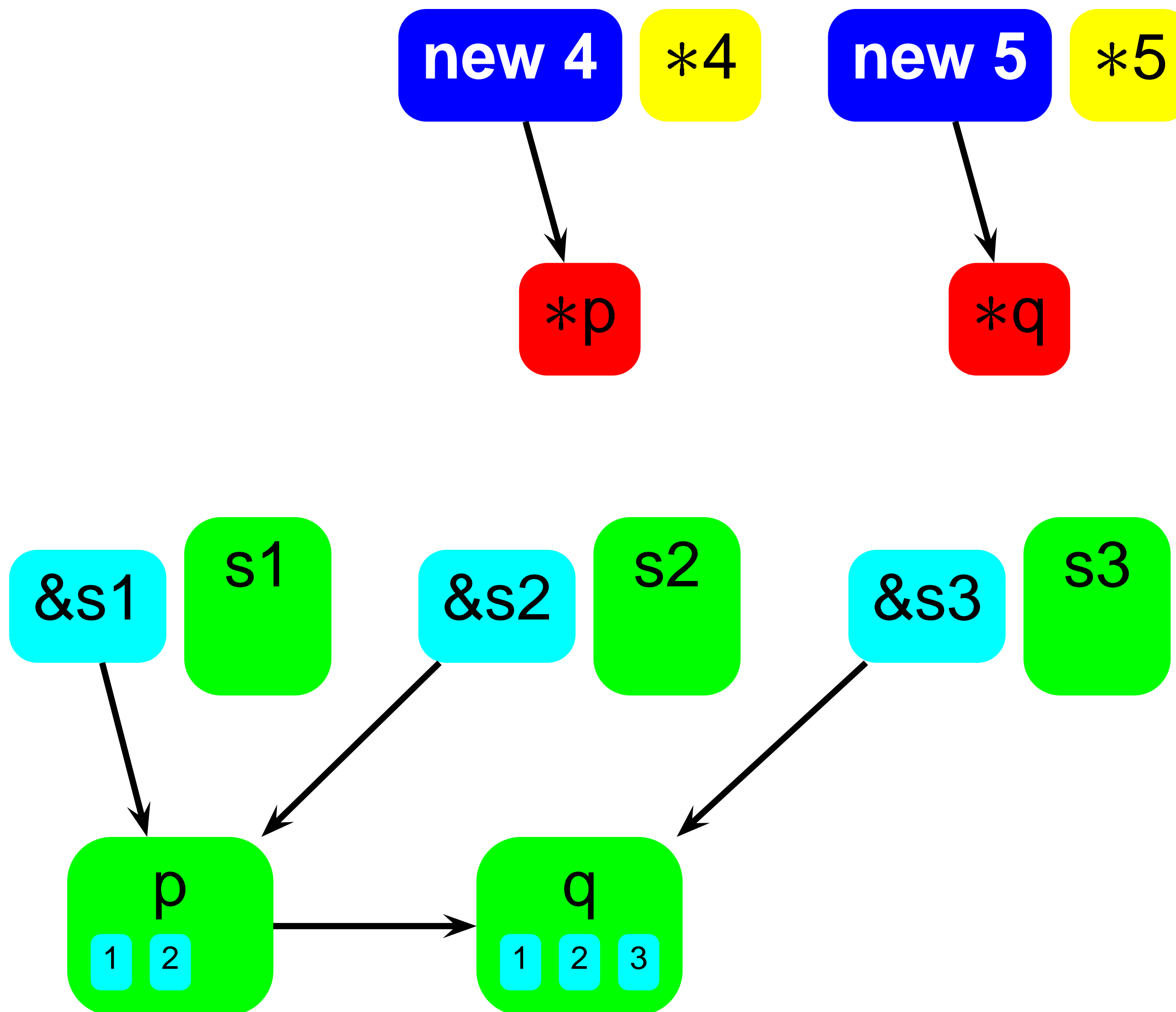
Andersen



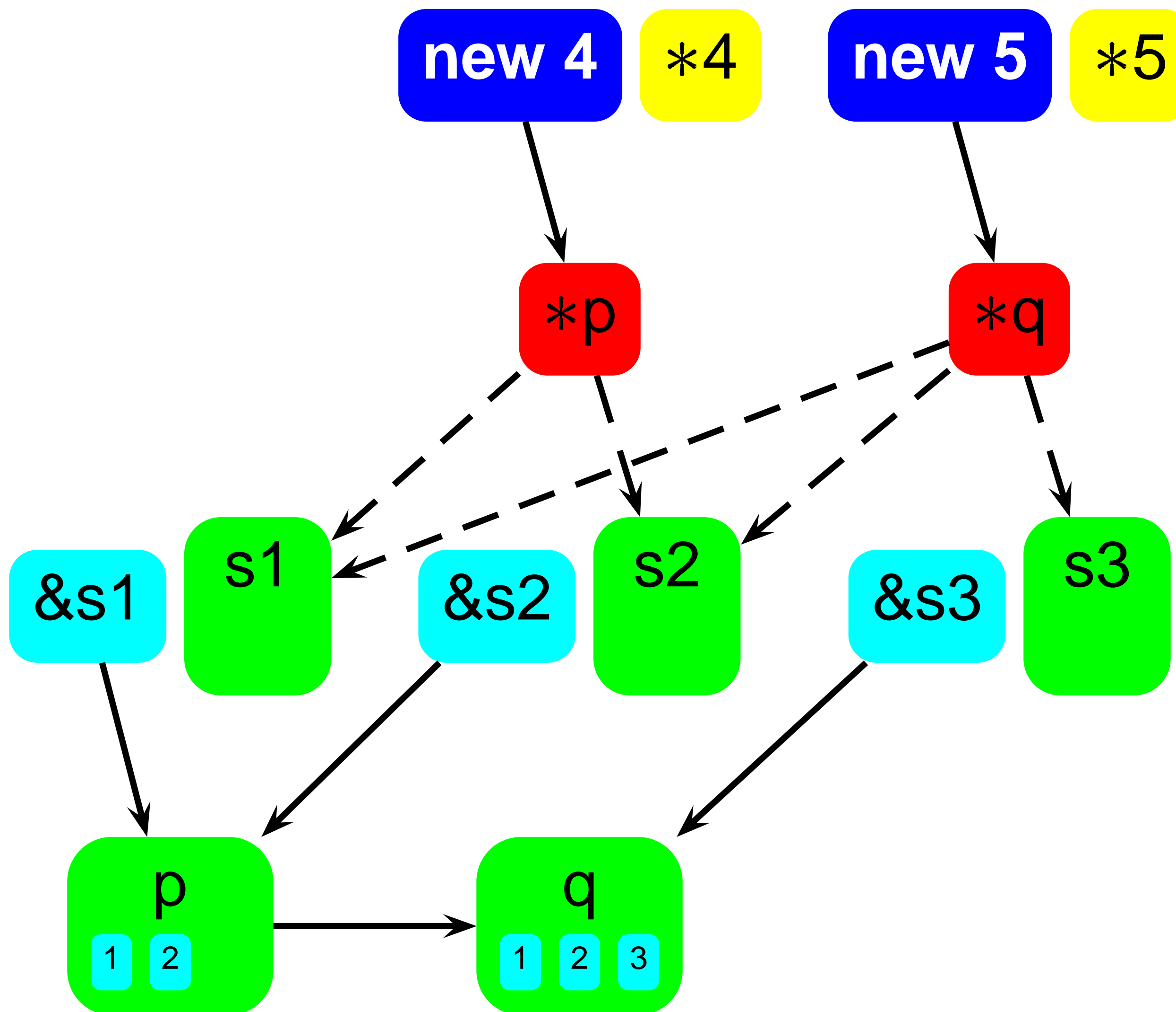
Andersen



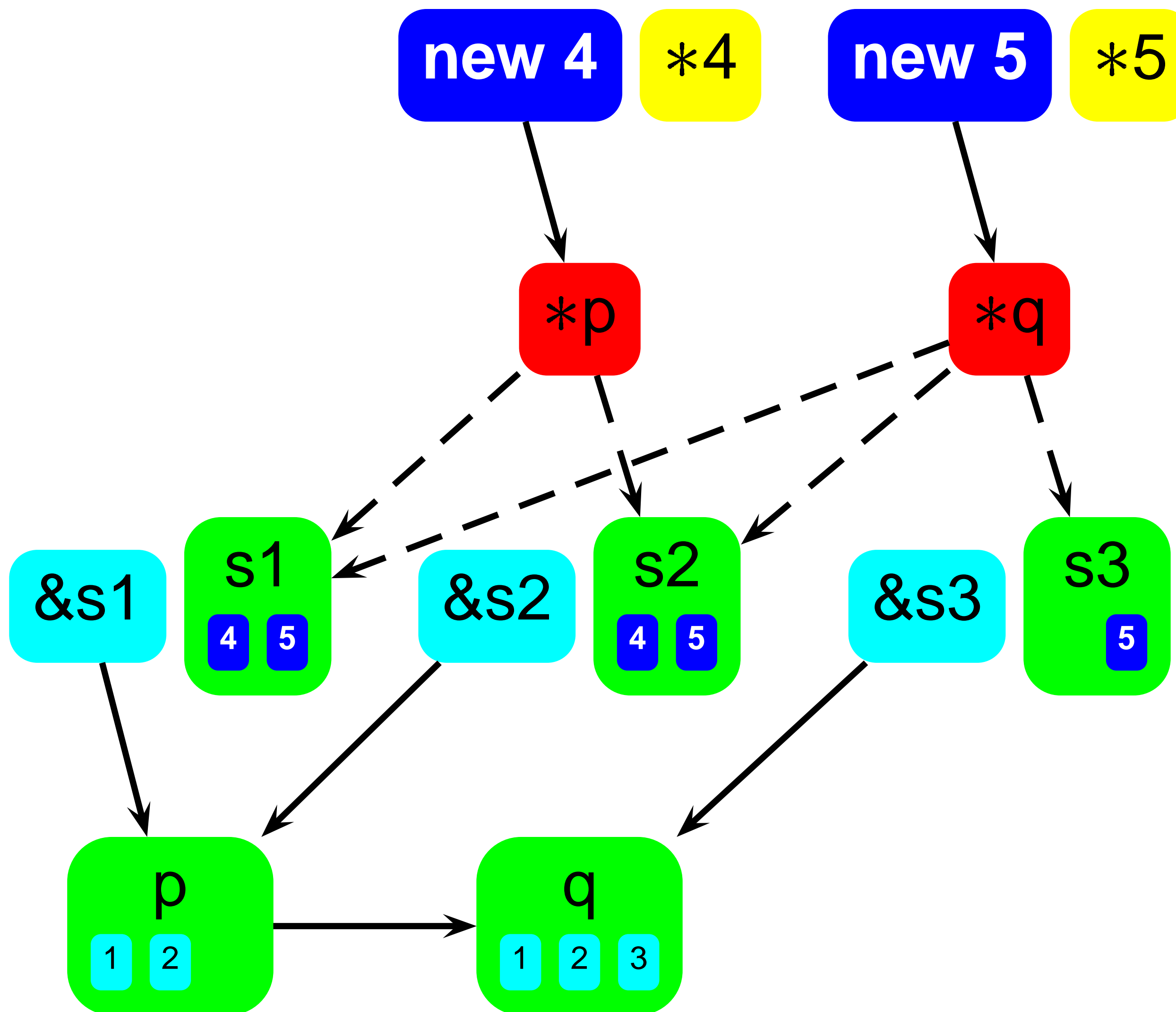
Andersen



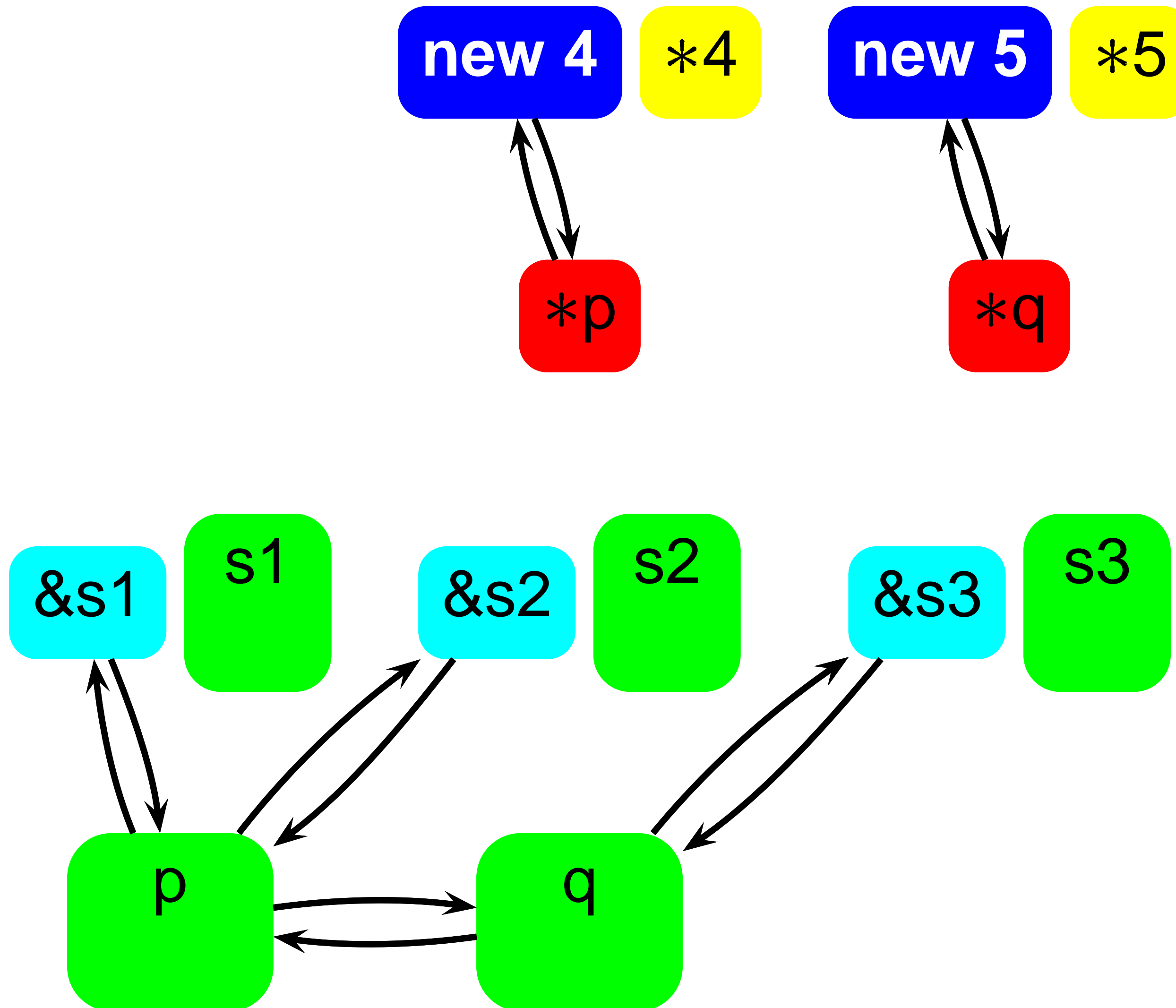
Andersen



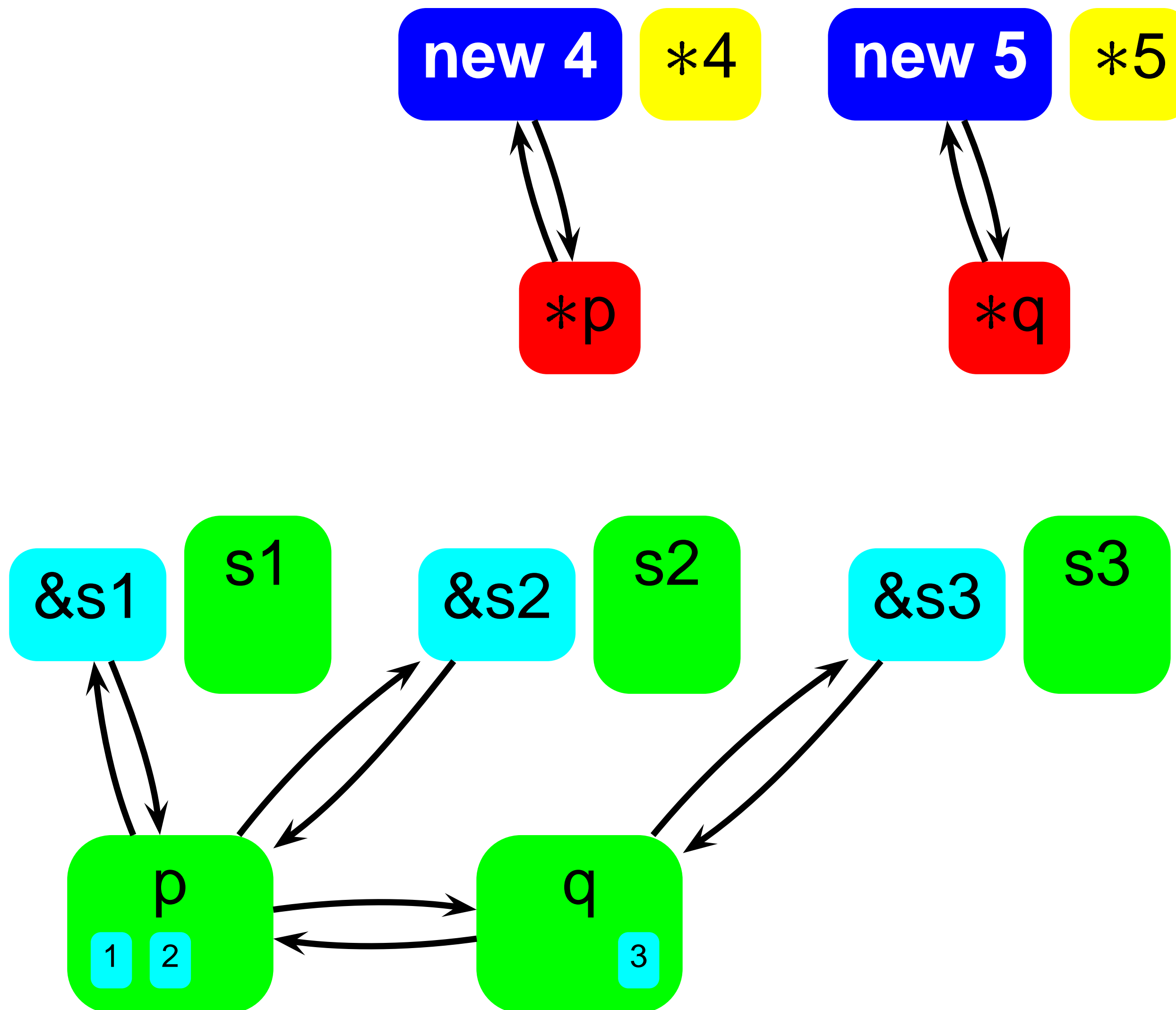
Andersen



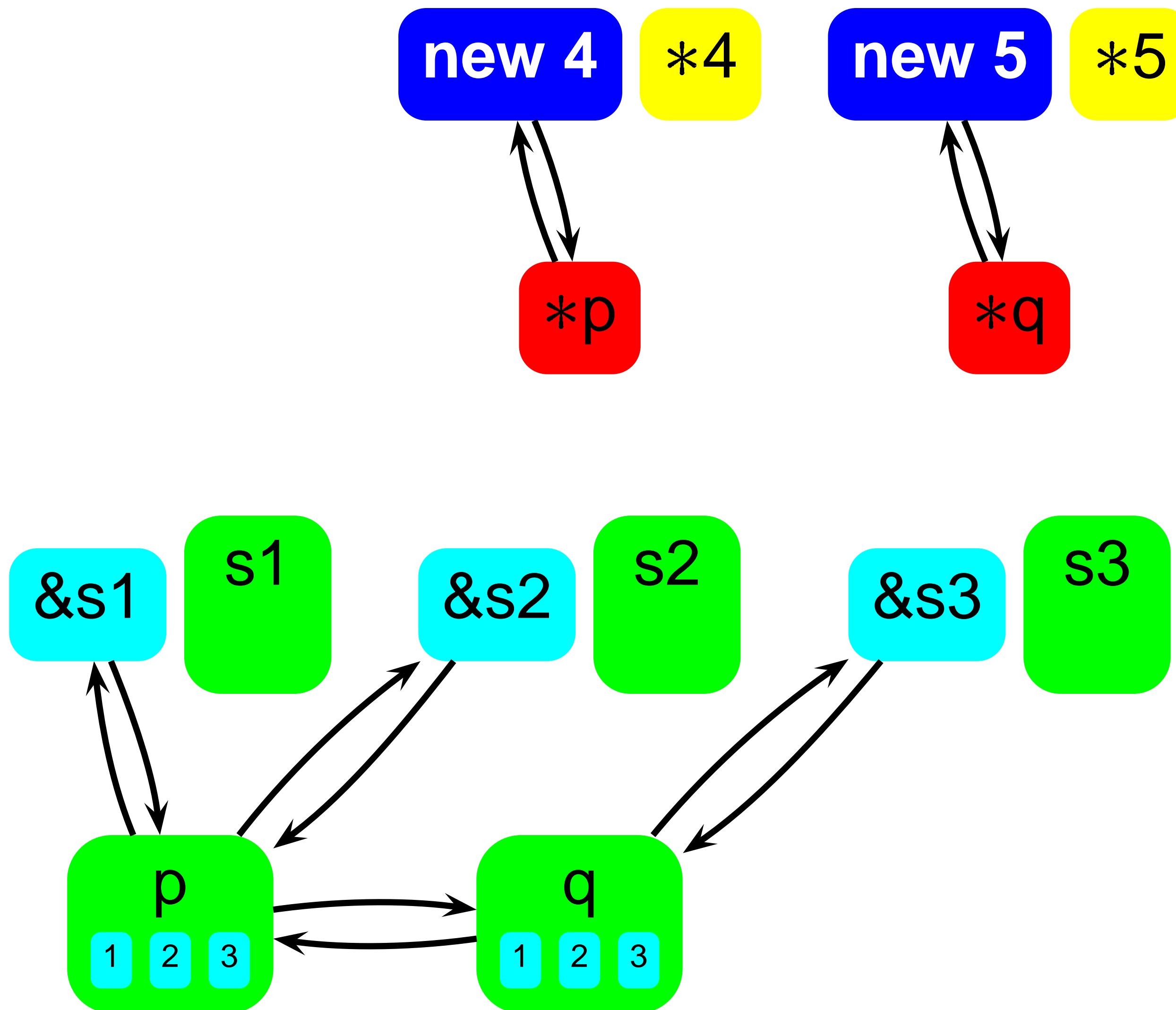
Steensgaard



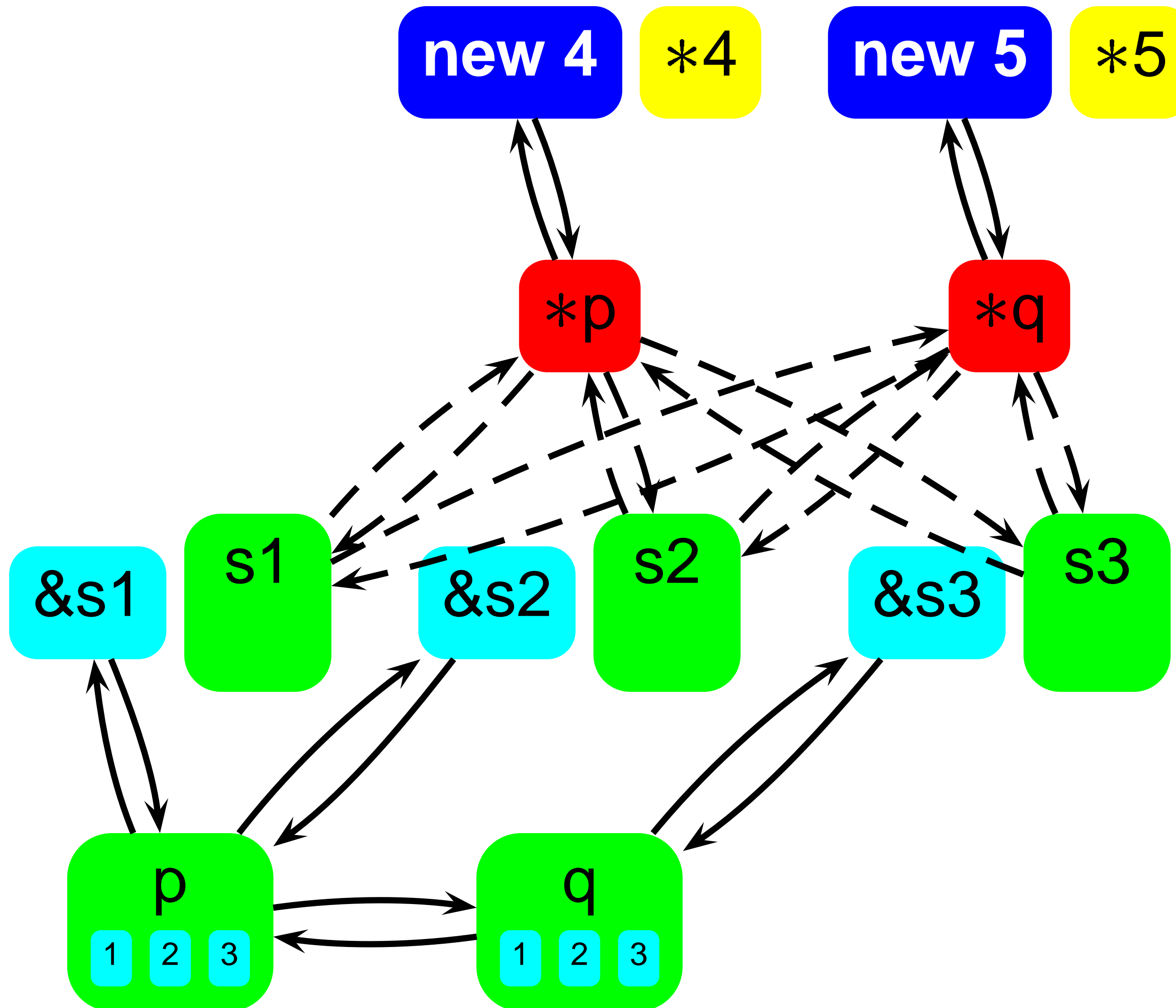
Steensgaard



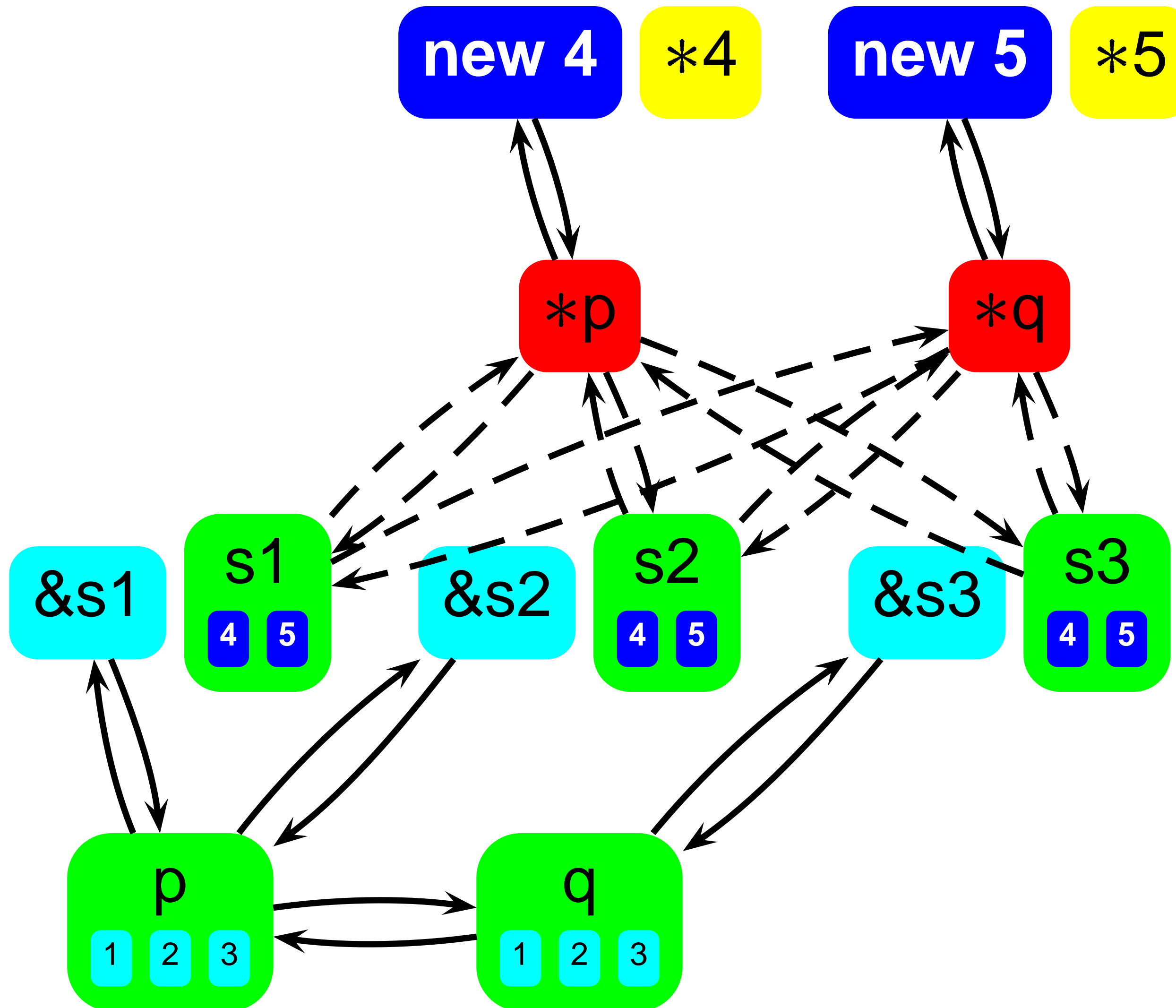
Steensgaard



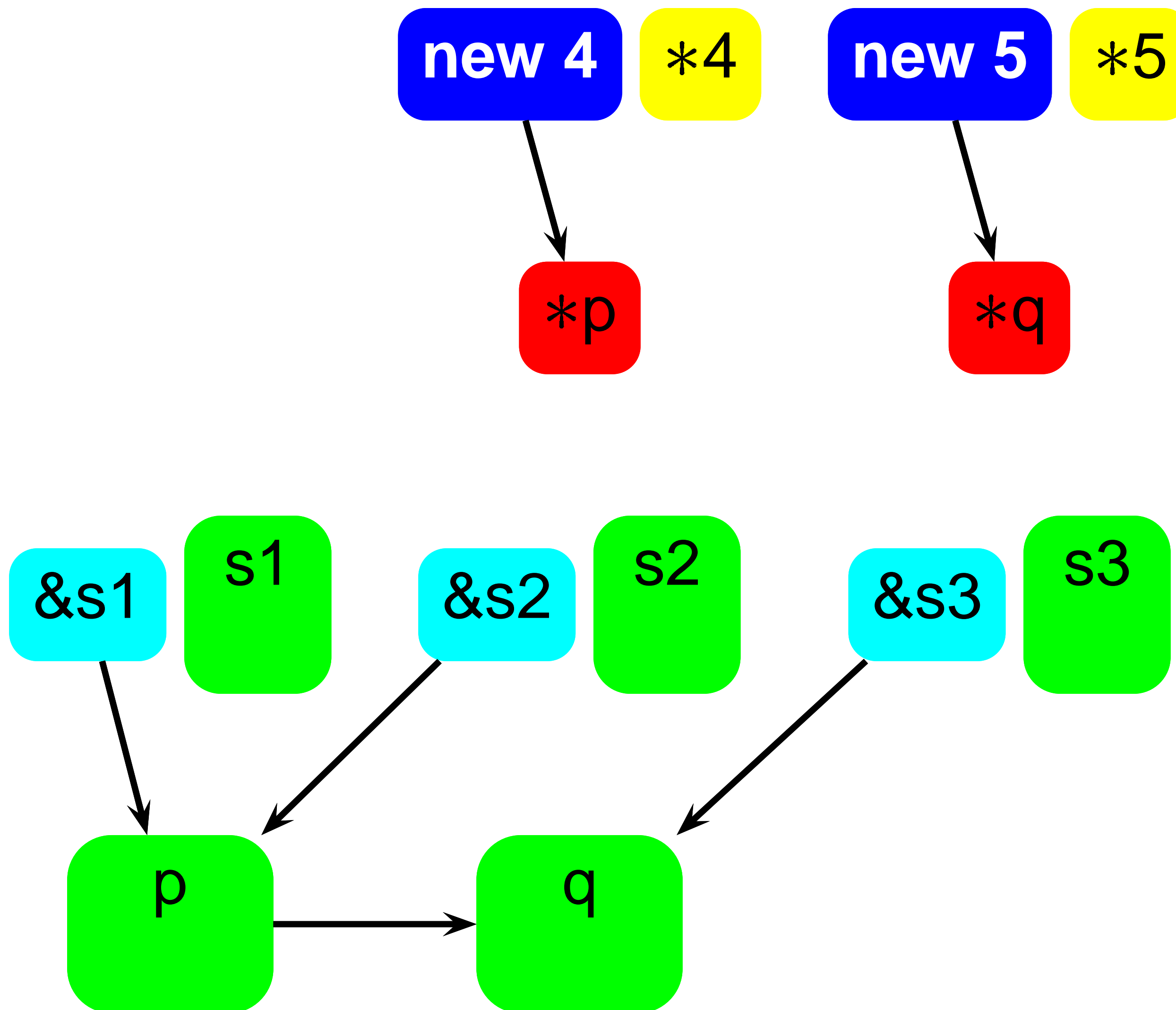
Steensgaard



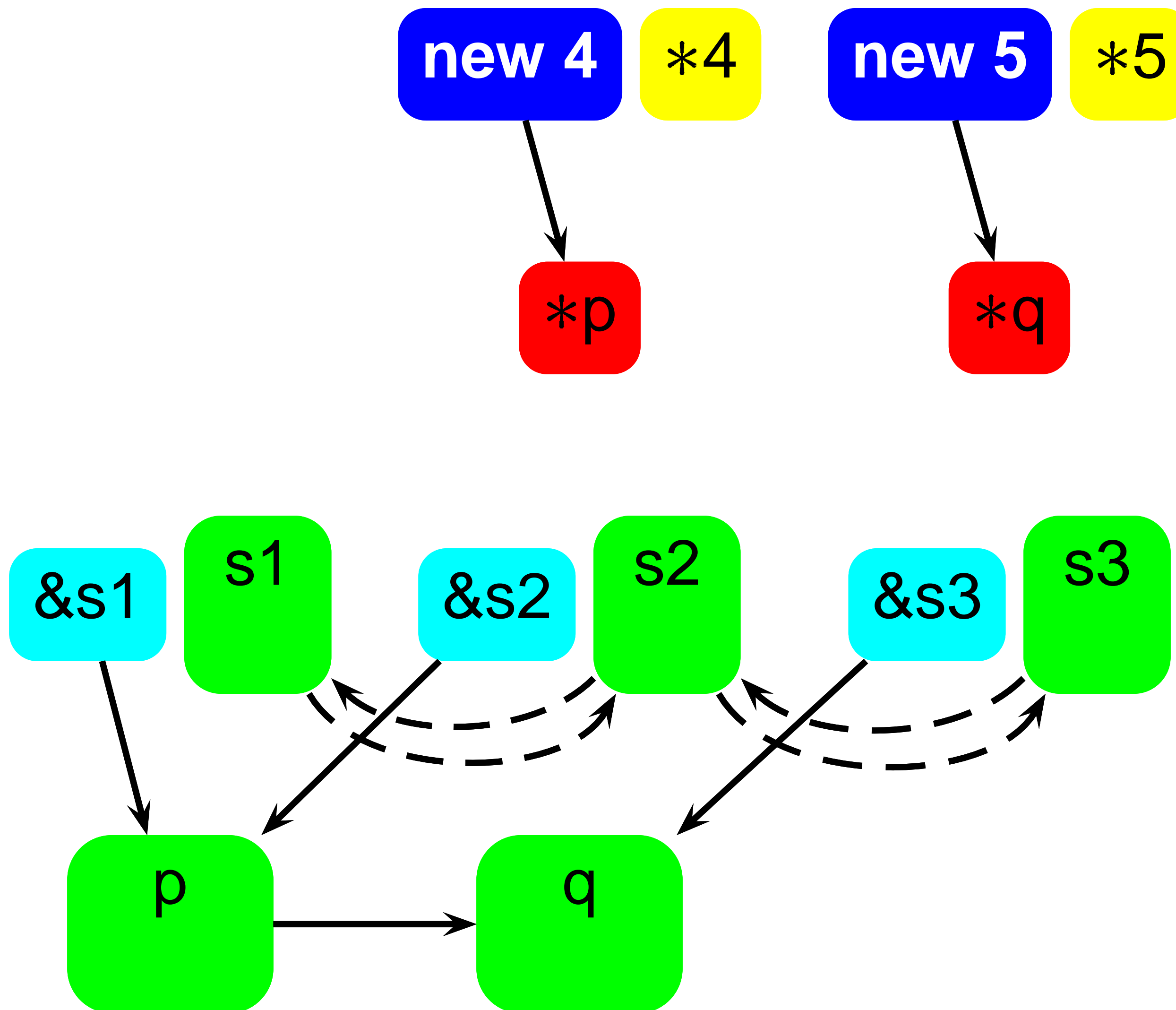
Steensgaard



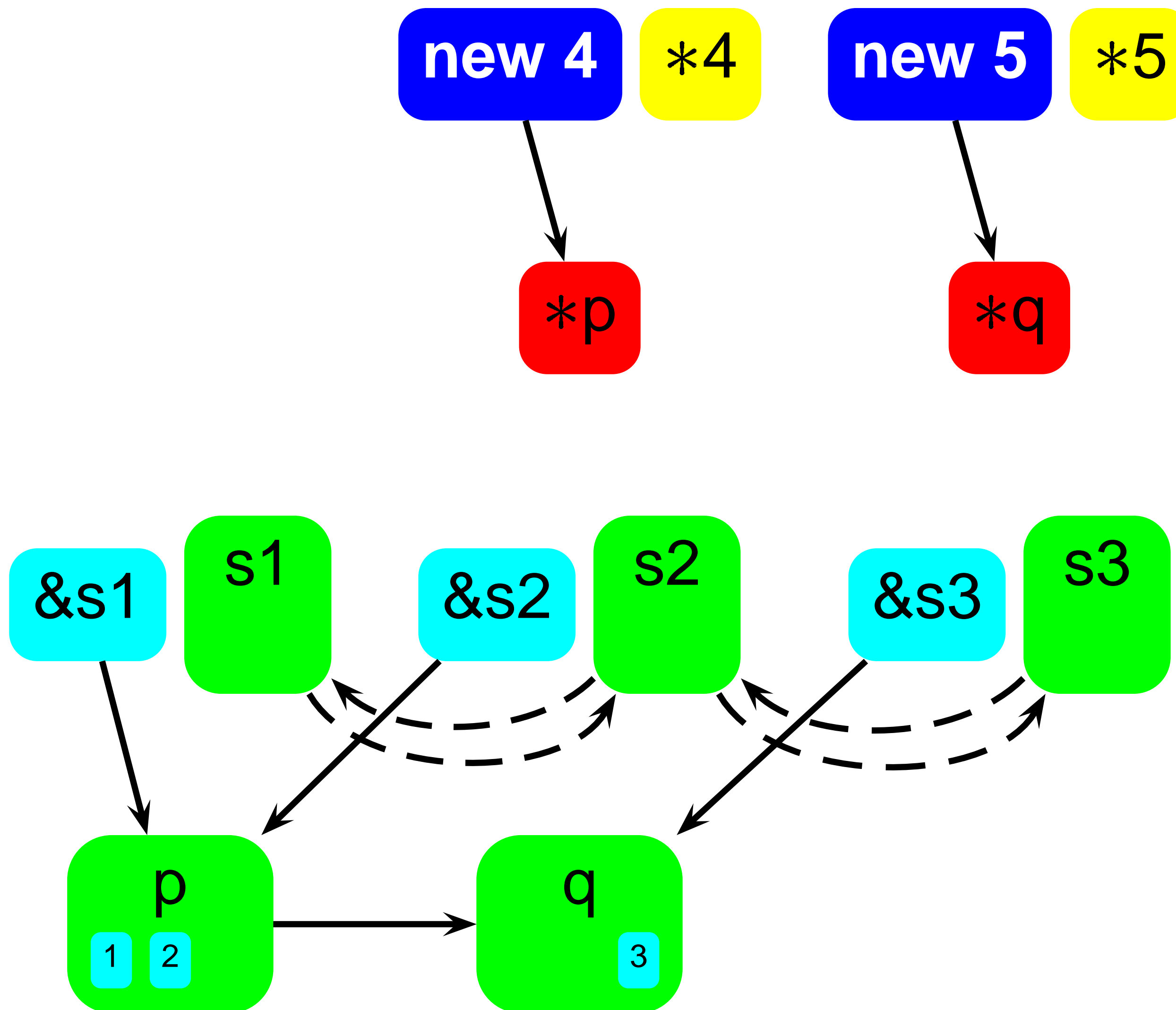
Das



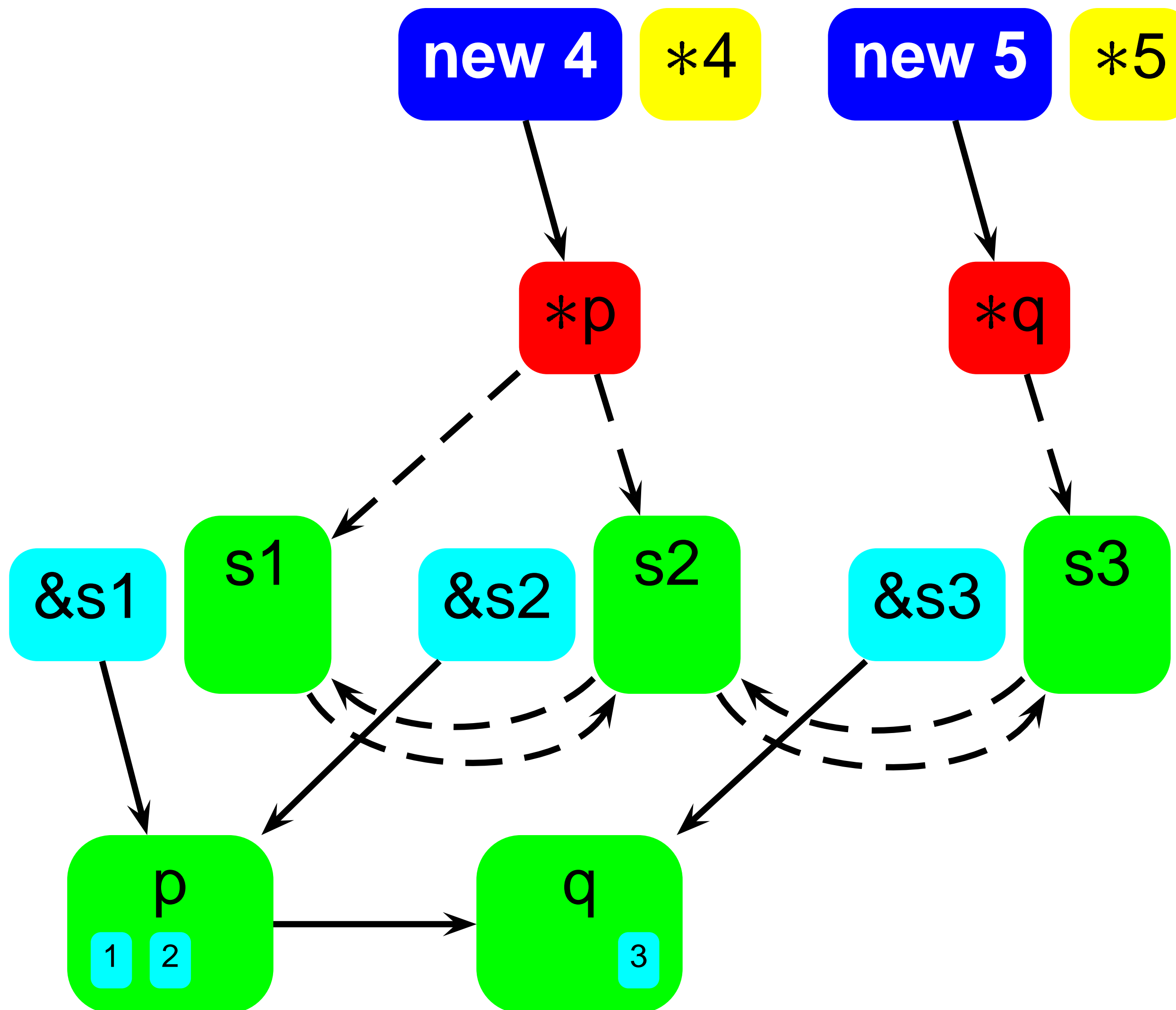
Das



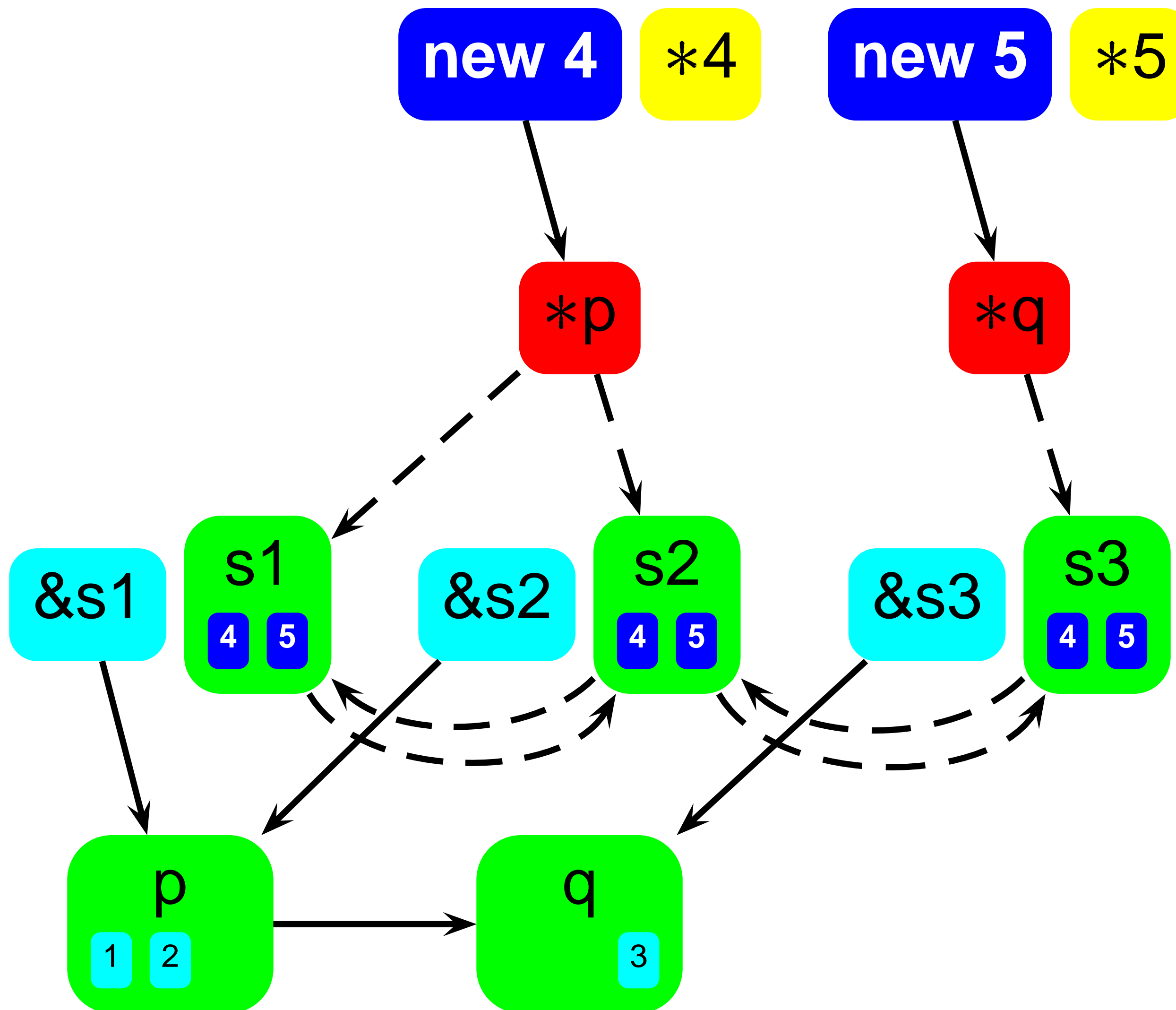
Das



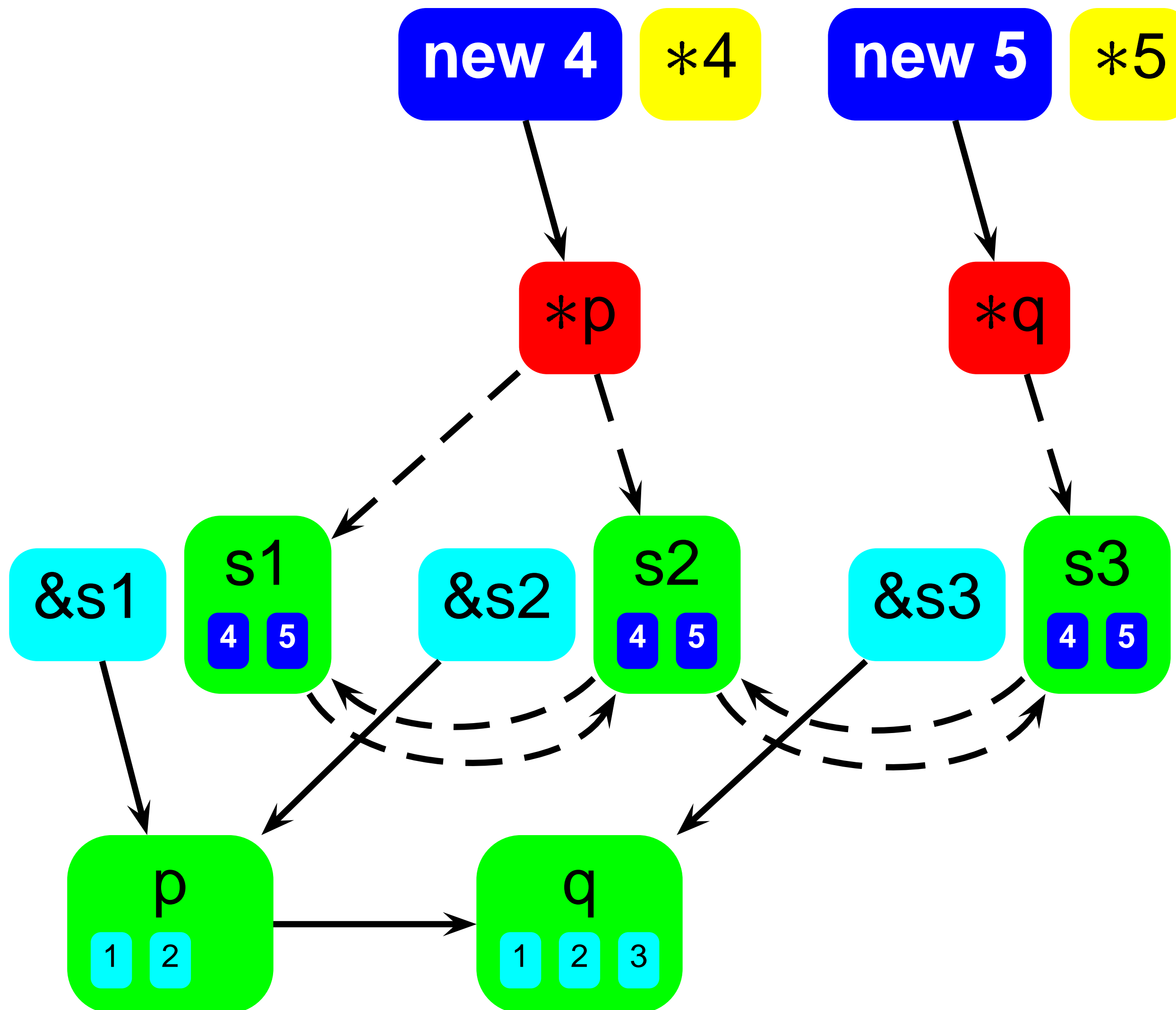
Das



Das



Das



Das

Program	Analysis time (secs)		Average thru-deref size		
	Ste96	Flow	Ste96	And94	Flow
compress	0.03	0.05	2.1	1.22	1.22
li	0.43	0.67	287.7	185.62	185.62
m88ksim	0.79	1.22	86.3	3.19	3.29
ijpeg	0.97	1.51	17.0	11.76	11.78
go	0.89	1.42	45.2	14.79	14.79
perl	1.21	2.12	36.1	22.22	22.22
vortex	3.35	5.66	1,064.5	45.54	59.30
gcc	5.70	9.45	245.8	7.71	7.72
Word97	61.34	126.83	27,258.6		11,219.5

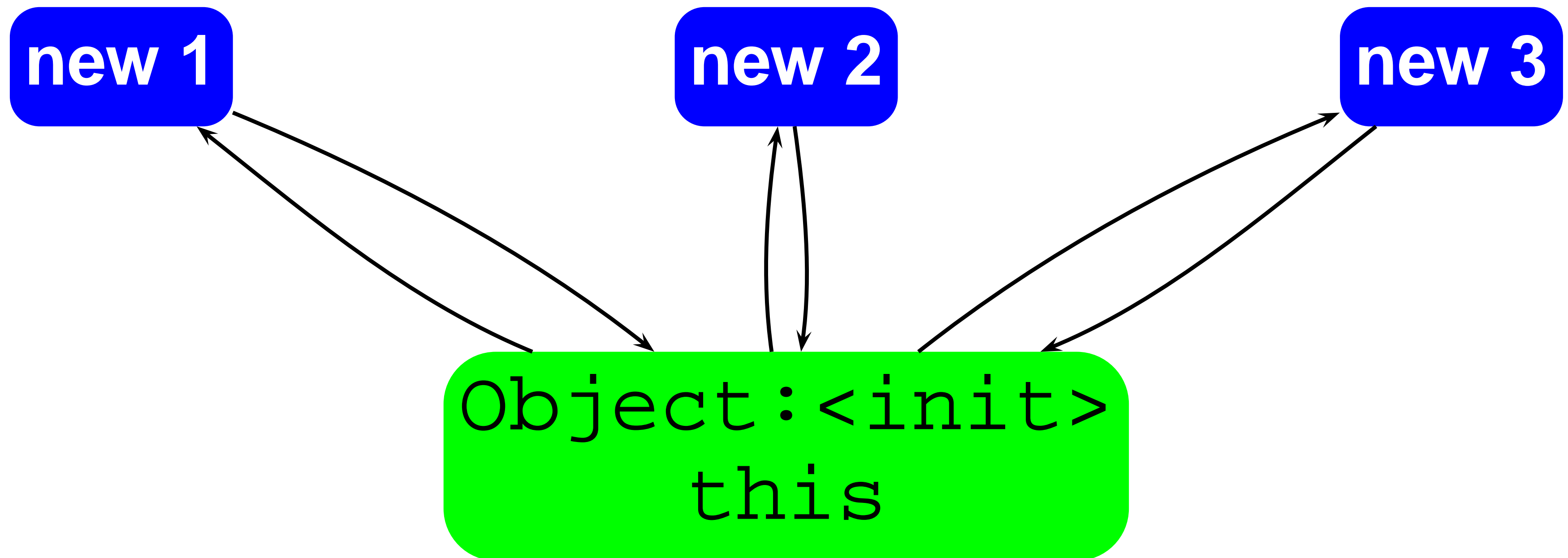
Liang, Pennings, Harrold

Issues in Java points-to analyses:

- Steensgaard vs. Andersen
- Fields
- Call graph
- Library
- Casts and declared types

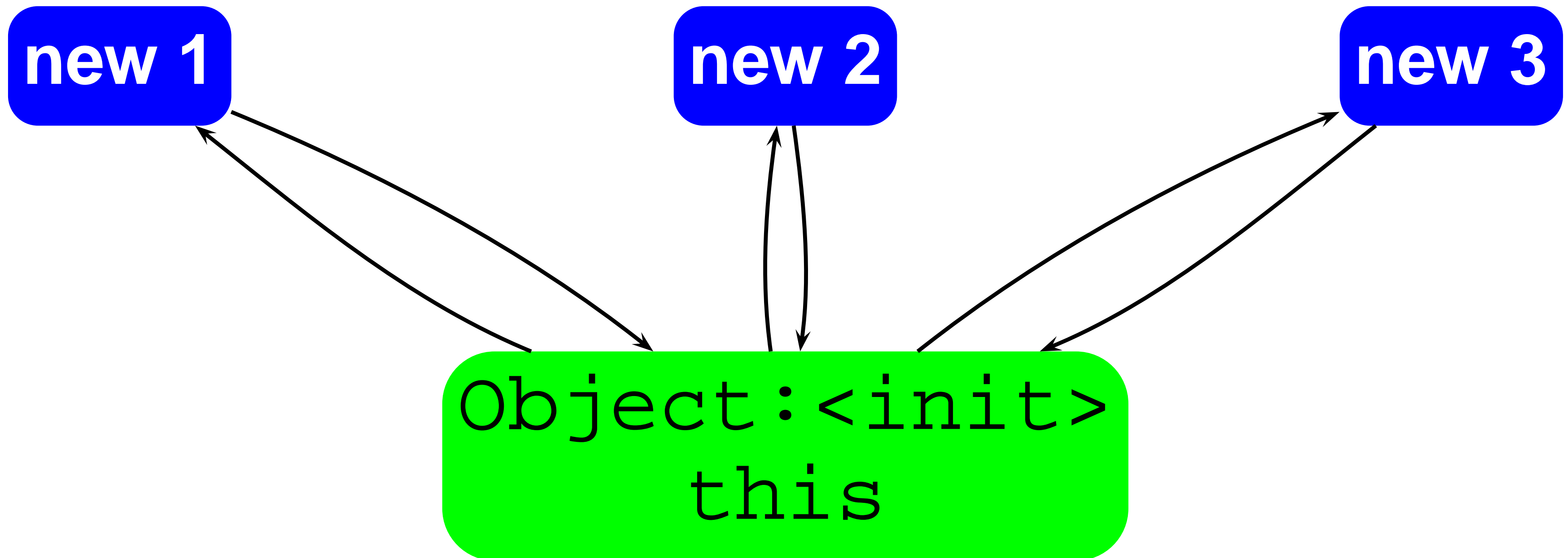
Liang, Pennings, Harrold

Problem with Steensgaard's analysis:



Liang, Pennings, Harrold

Problem with Steensgaard's analysis:

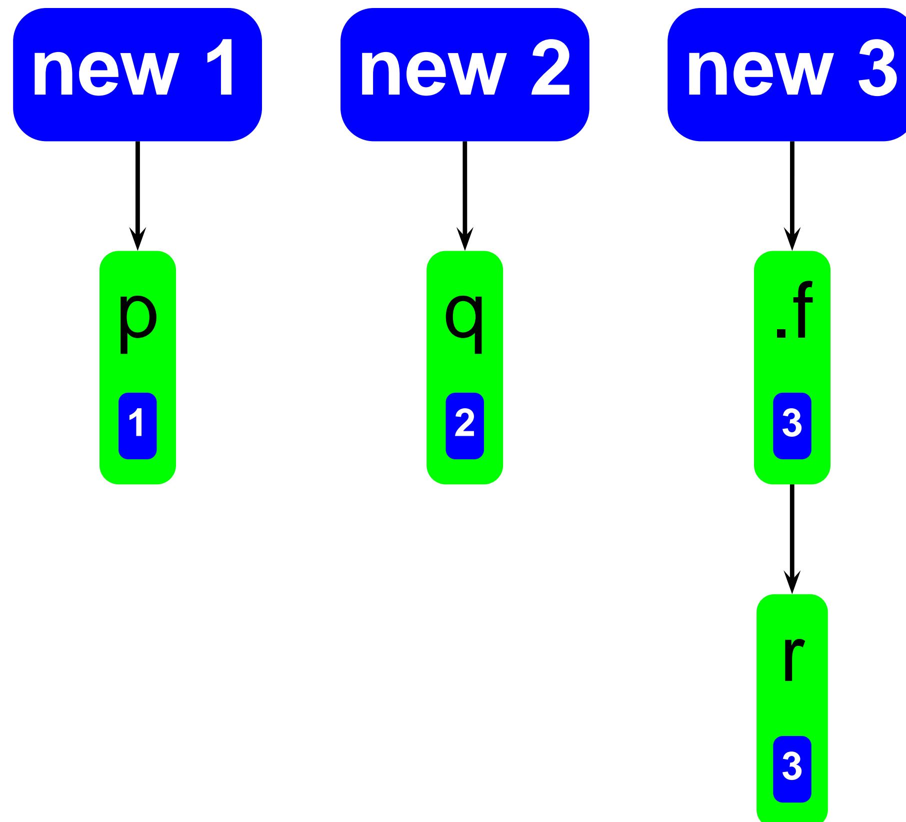


Solution: remove `this` from the graph unless it is needed (assigned or dereferenced).

Fields

Option 1: Represent as variables (green nodes)

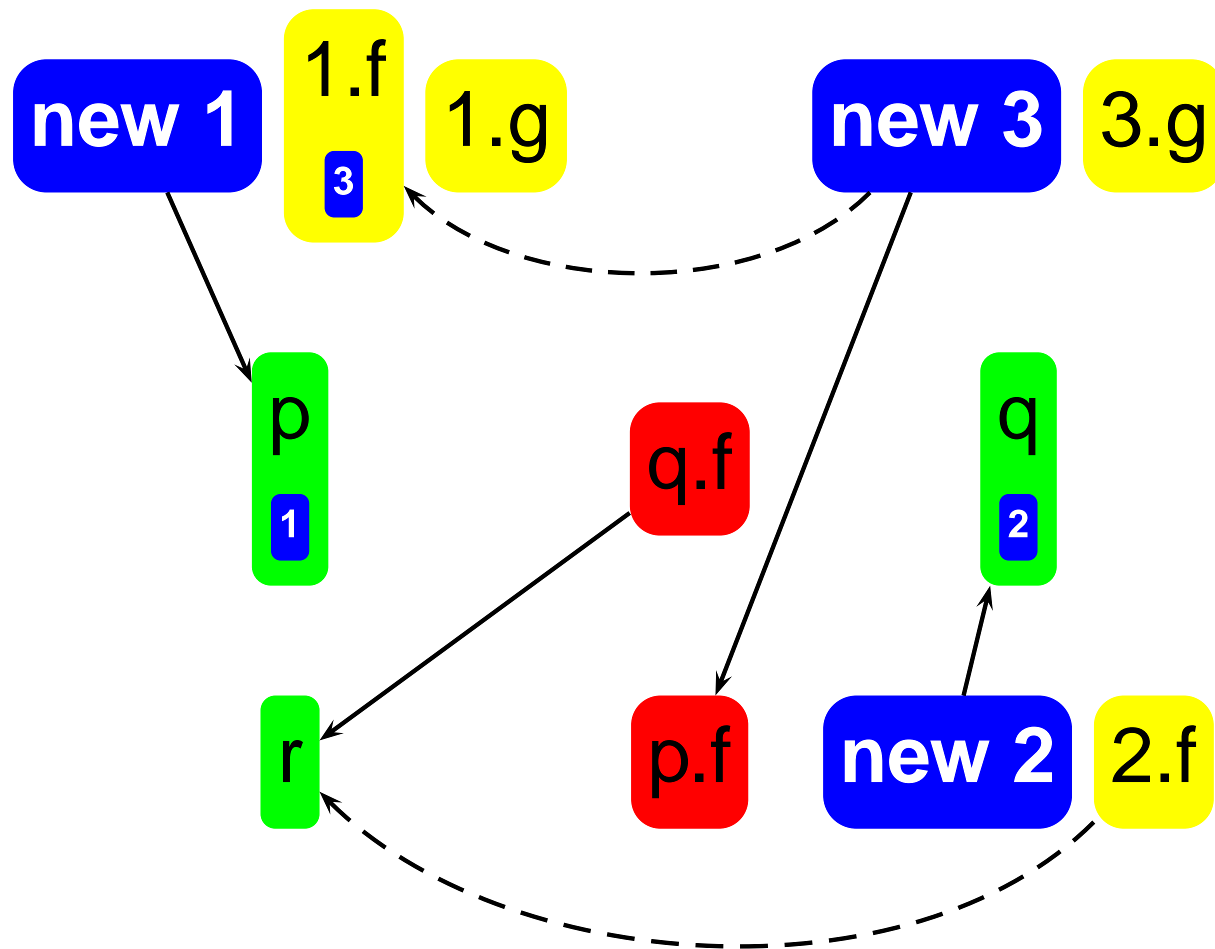
```
p = new 1;
q = new 2;
p.f = new 3;
r = q.f;
```



Fields

Option 2: Represent as objects (yellow nodes)

```
p = new 1;
q = new 2;
p.f = new 3;
r = q.f;
```



Fields

Option 1:

- No dashed lines $\longrightarrow$ no iteration
- Only blue and green nodes $\longrightarrow$ simpler

Option 2:

- More precise (r doesn't get 3)

Call Graph

- CHA

Call Graph

- CHA
- RTA

Call Graph

- CHA
- RTA
- On-the-fly

Call Graph

- CHA
- RTA
- On-the-fly
  - Most precise but more complicated

Call Graph

- CHA
- RTA
- On-the-fly
  - Most precise but more complicated
  - More iteration

Library

- Collections simulated as arrays

Library

- Collections simulated as arrays
- Faster

Library

- Collections simulated as arrays
- Faster
- More precise

Library

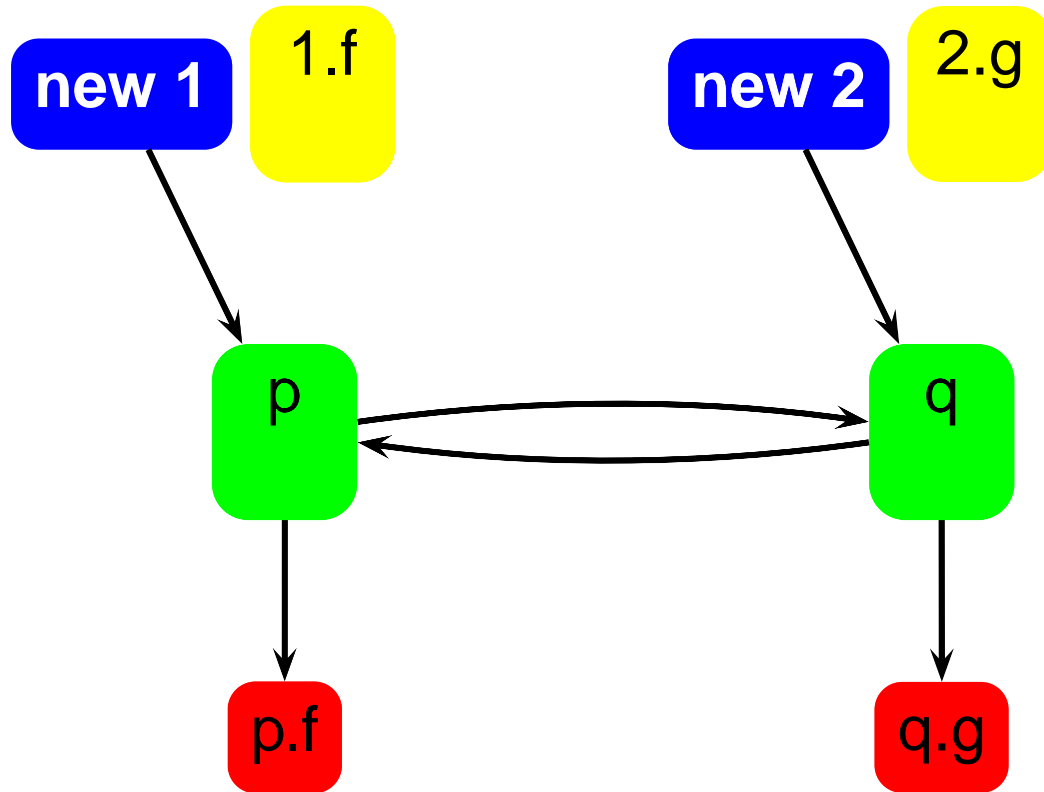
- Collections simulated as arrays
- Faster
- More precise
- Lots of work, error-prone

Casts and declared types

- Each location (green or yellow) has declared type
- Can only hold references (blue or cyan) of compatible actual type
- Difficult if nodes are merged (Steensgaard)
- Improves precision
- Reduces graph growth if fields represented as objects

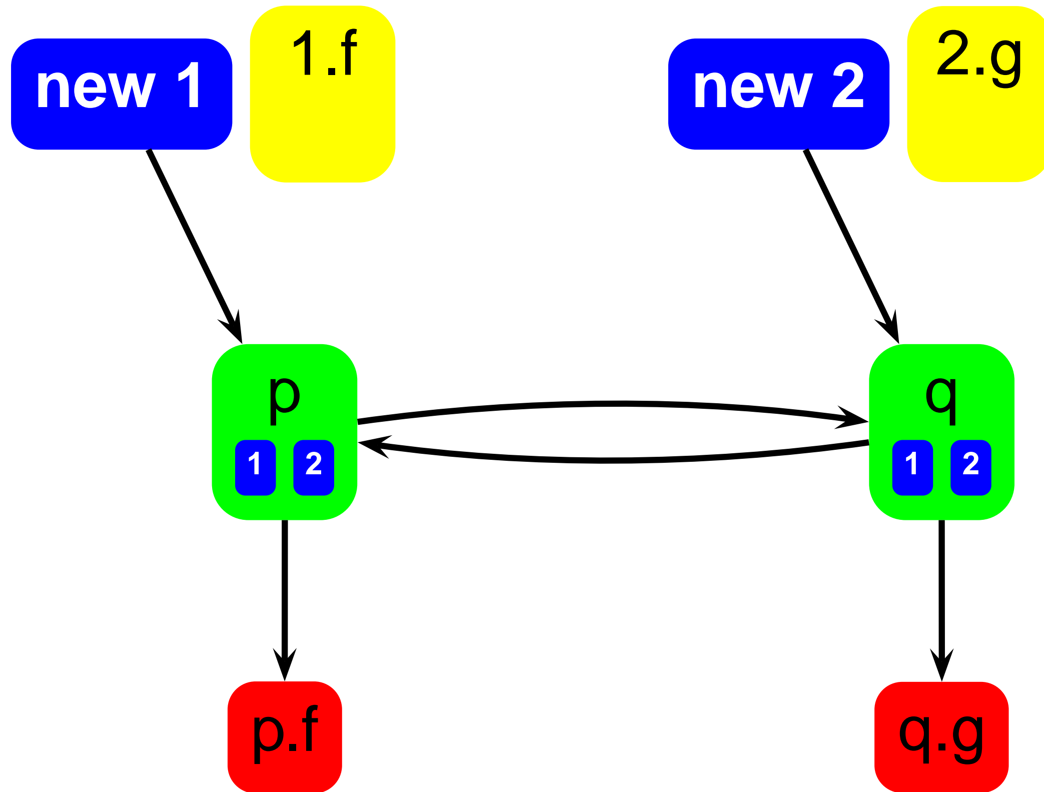
Casts and declared types

```
p = new 1;
q = new 2;
p = q;
q = p;
p.f = p;
q.g = q;
```



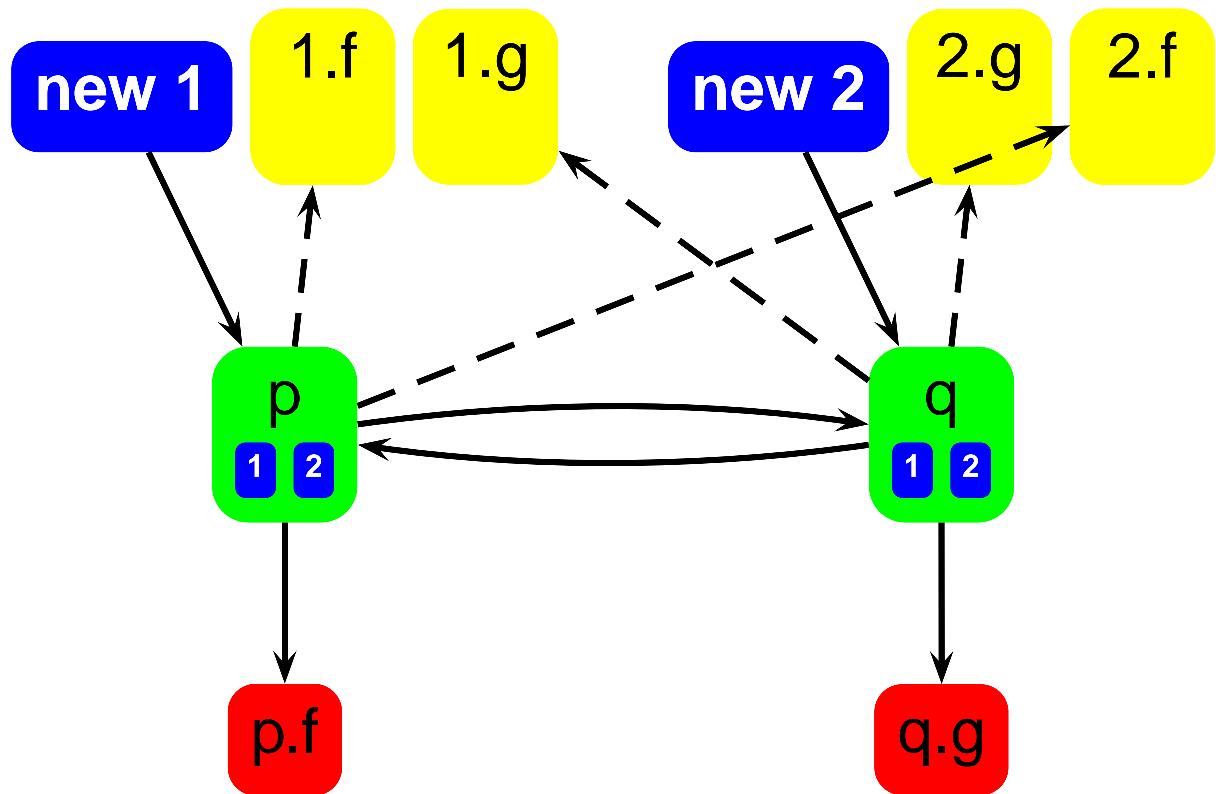
Casts and declared types

```
p = new 1;
q = new 2;
p = q;
q = p;
p.f = p;
q.g = q;
```



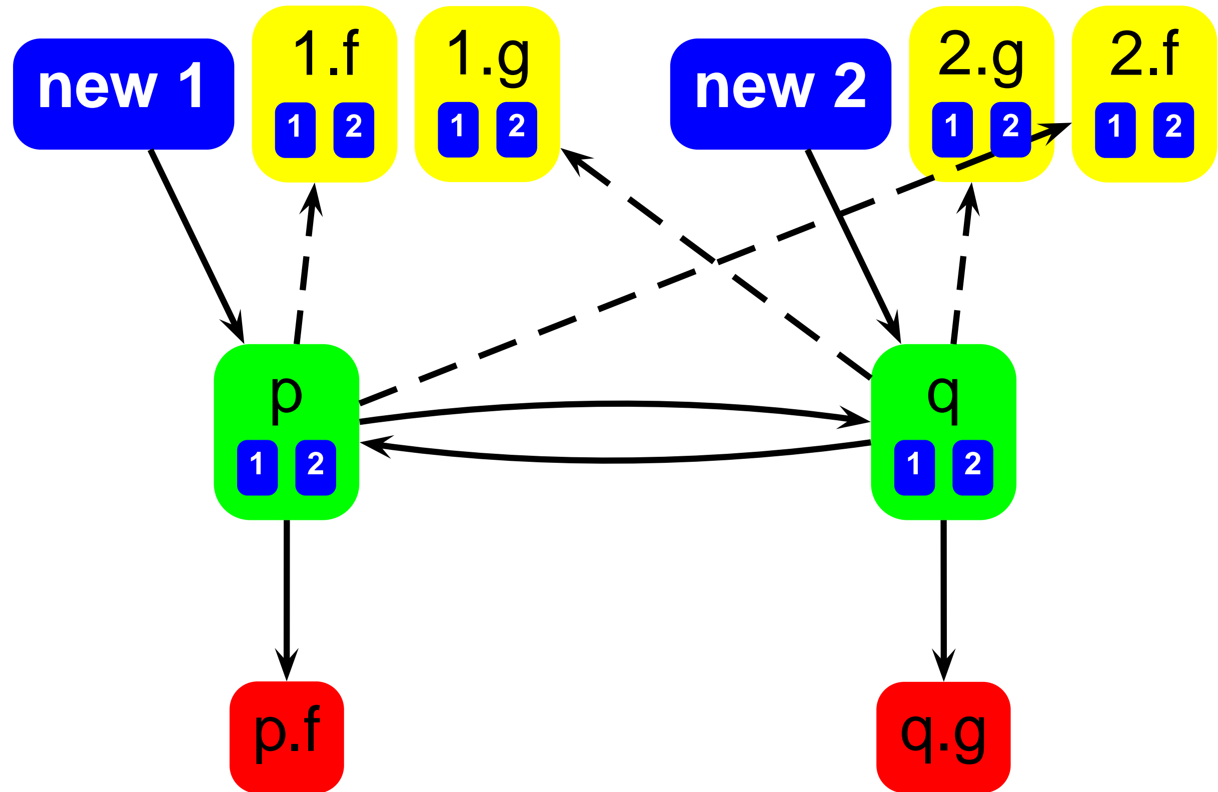
Casts and declared types

```
p = new 1;
q = new 2;
p = q;
q = p;
p.f = p;
q.g = q;
```

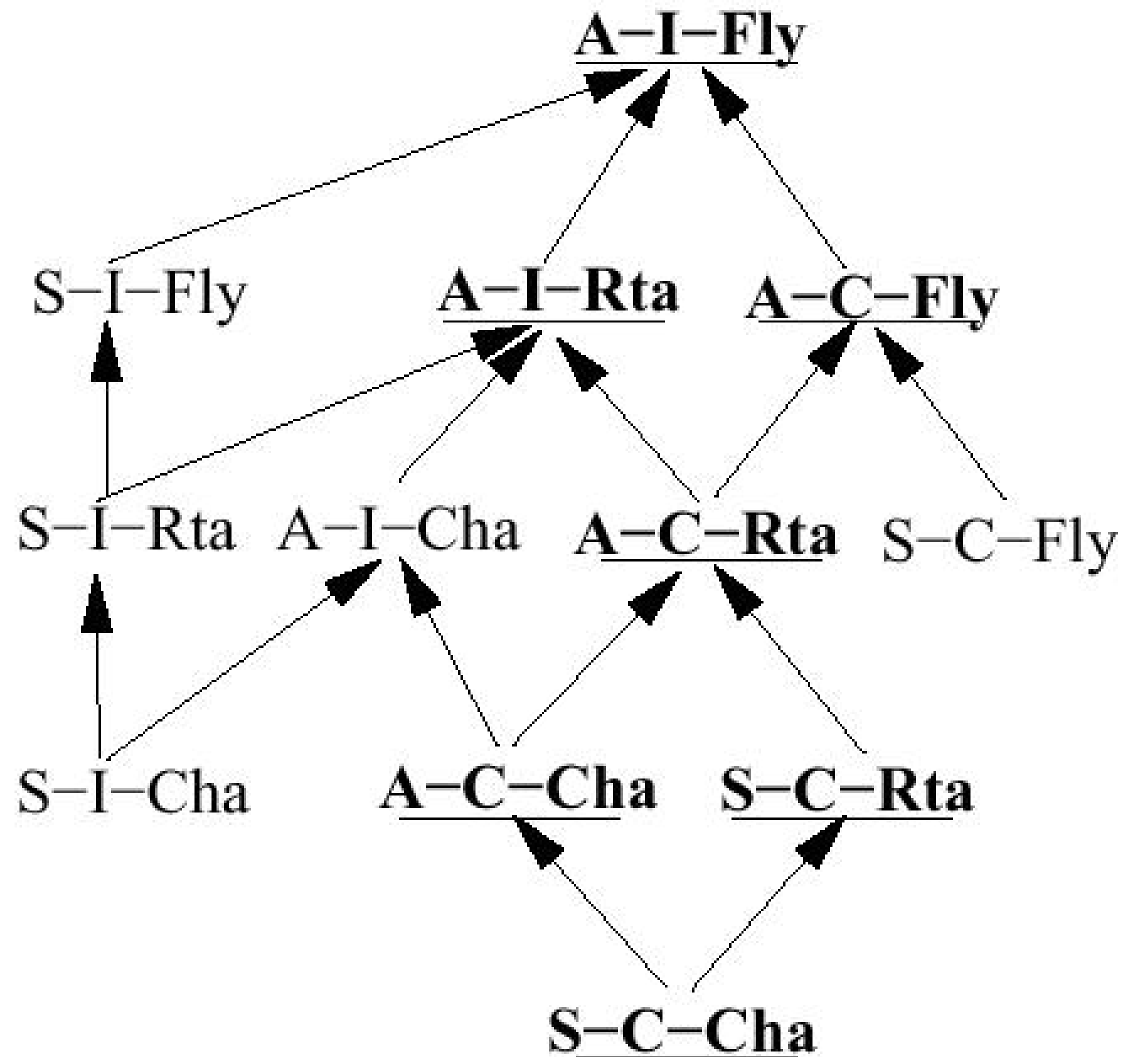


Casts and declared types

```
p = new 1;
q = new 2;
p = q;
q = p;
p.f = p;
q.g = q;
```



Liang, Pennings, Harrold



Liang, Pennings, Harrold

- Steensgaard's with `this` trick better than nothing

Liang, Pennings, Harrold

- Steensgaard's with `this` trick better than nothing
- Andersen's: fields as objects made almost no difference in precision for virtual call resolution and escape analysis

Liang, Pennings, Harrold

- Steensgaard's with `this` trick better than nothing
- Andersen's: fields as objects made almost no difference in precision for virtual call resolution and escape analysis
- Fields as objects up to 5 times slower (but they don't give implementation details. . .)

Conclusions

- Flow graph can represent various points-to analyses
- Steensgaard's analysis is fast but imprecise
- Andersen's analysis is precise but slow
- In-between analyses can be fast and precise
- Java is not exactly like C