Heap Analysis with Collections

Mark Marron

Deepak Kapur

Darko Stefanovic

Manuel Hermenegildo

University of New Mexico

marron@cs.unm.edu

PASTE 2007

Shape Analysis Results

- Aliasing
- Connectivity
  - Reachability
  - Interference
  - Paths
- Logical data structures (Regions)
  - Group related sections of the heap
  - Keep unrelated sections of the heap separate
- Shape of a region
  - Cycle, Dag, Tree, ***List***, ***Singleton***

Why Collections?

Java, C++/STL provide a wide range of standard collections that are used in almost any non-trivial program.

Modeling Collections is Hard

- Collection implementations are often complex and difficult to analyze
  - Some techniques are efficient but are unable to accurately model the complexities of the implementation
  - Some techniques can analyze all the complexities of the implmentations but are computationally expensive

Proposed Approach

- Model the collections using abstracted semantics
- Represent all the pointers stored in a collection via a summary representation
- When accessing the contents of the collection make the specific element being accessed explicit
- Use the concept of Iterators to define an order for the collection so we can track progress of loops etc.

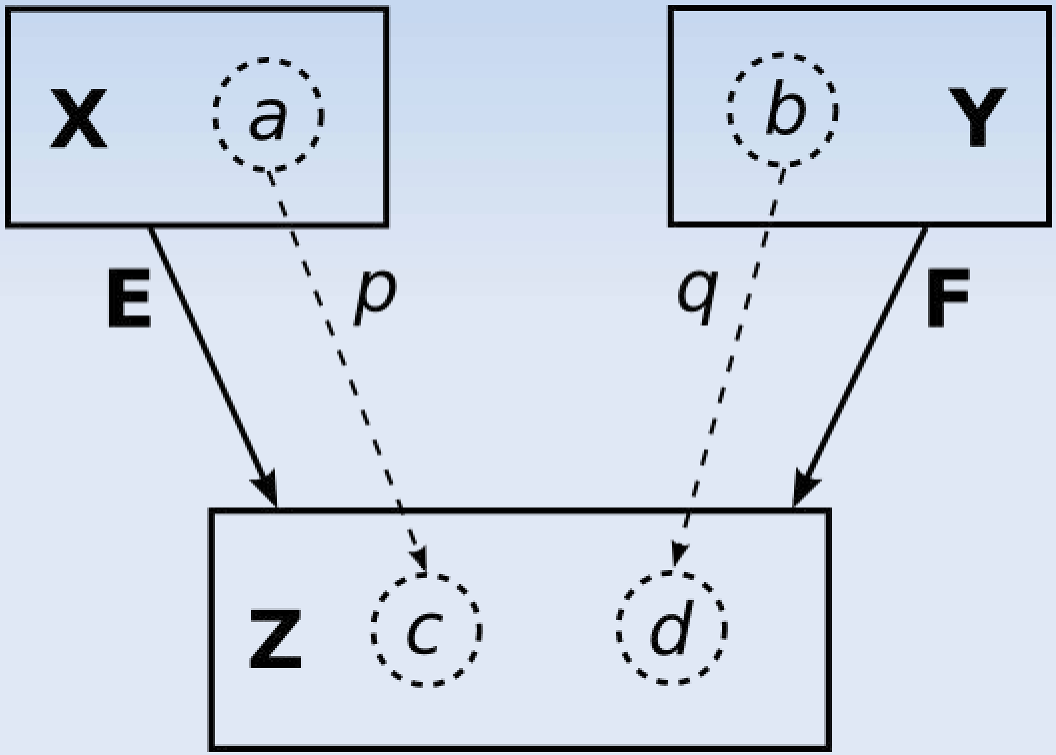
Abstract Domain

- Classic heap graph model
  - Nodes represent regions (sets of objects)
  - Edges represent sets of pointers
- Add extra abstract properties to nodes
  - Types of the concrete memory represented
  - Total size
  - Internal structure
- Add extra abstract properties to the edges
  - Max number of parallel pointers
  - Potential that pointers interfere

Internal Structure

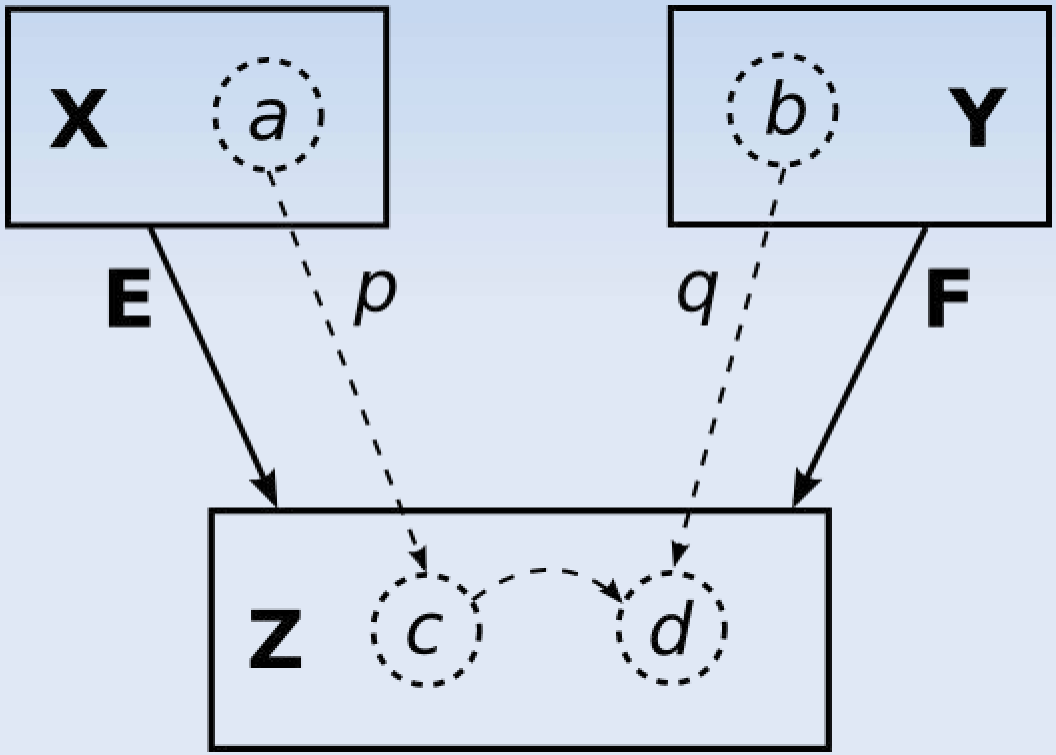
- Connectivity of incident edges
  - Do two incident edges represent pointers that refer to the same memory location or to connected memory locations?
- Internal layout
  - What is the “Shape” of the concrete memory locations that this node represents?

Connectivity



Memory locations c and d are ***disjoint***.
Edges E, F are ***disjoint***.
Region Z has a ***singleton*** layout shape.

Connectivity

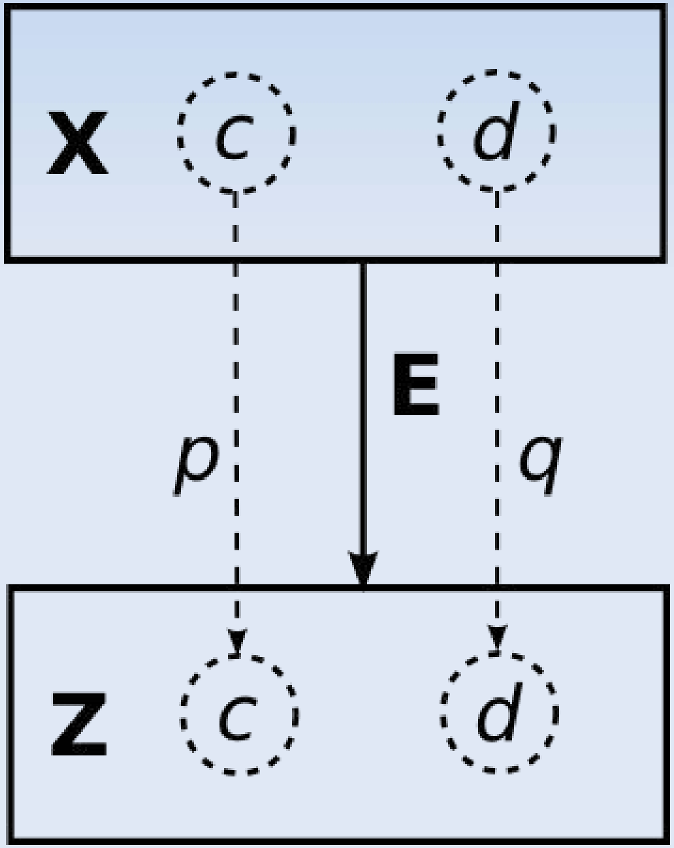


Memory locations c and d are ***connected***.
Edges E, F are ***connected***.
Region Z has a ***list*** layout shape.

Interference

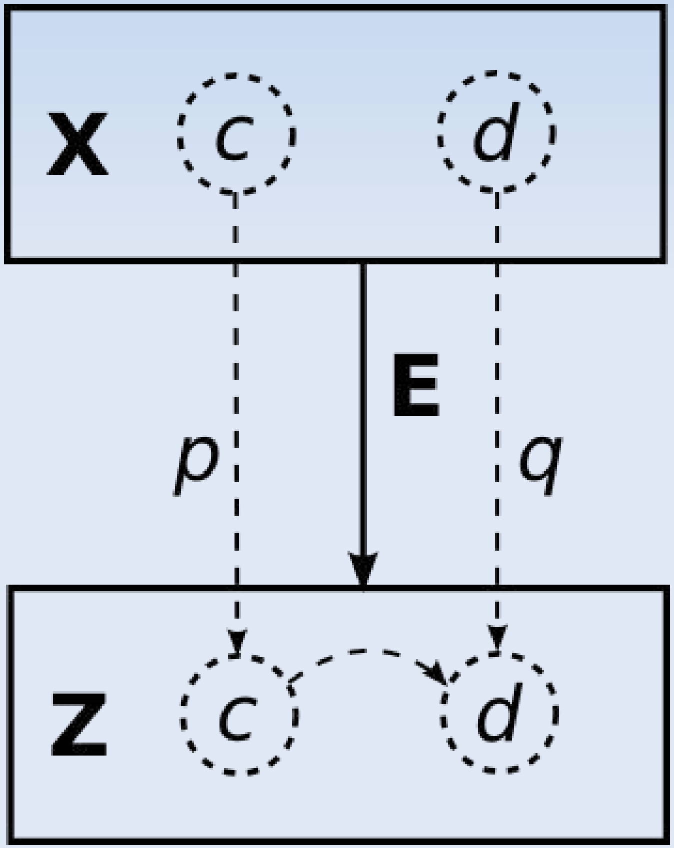
- Connectivity tracked the possibility that two distinct edges represent pointers that are connected.
- Interference tracks the possibility that two pointers represented by the same edge are connected.

Interference



Memory locations c and d are ***disjoint***.
Edge E is ***non-interfering (np)***.

Interference

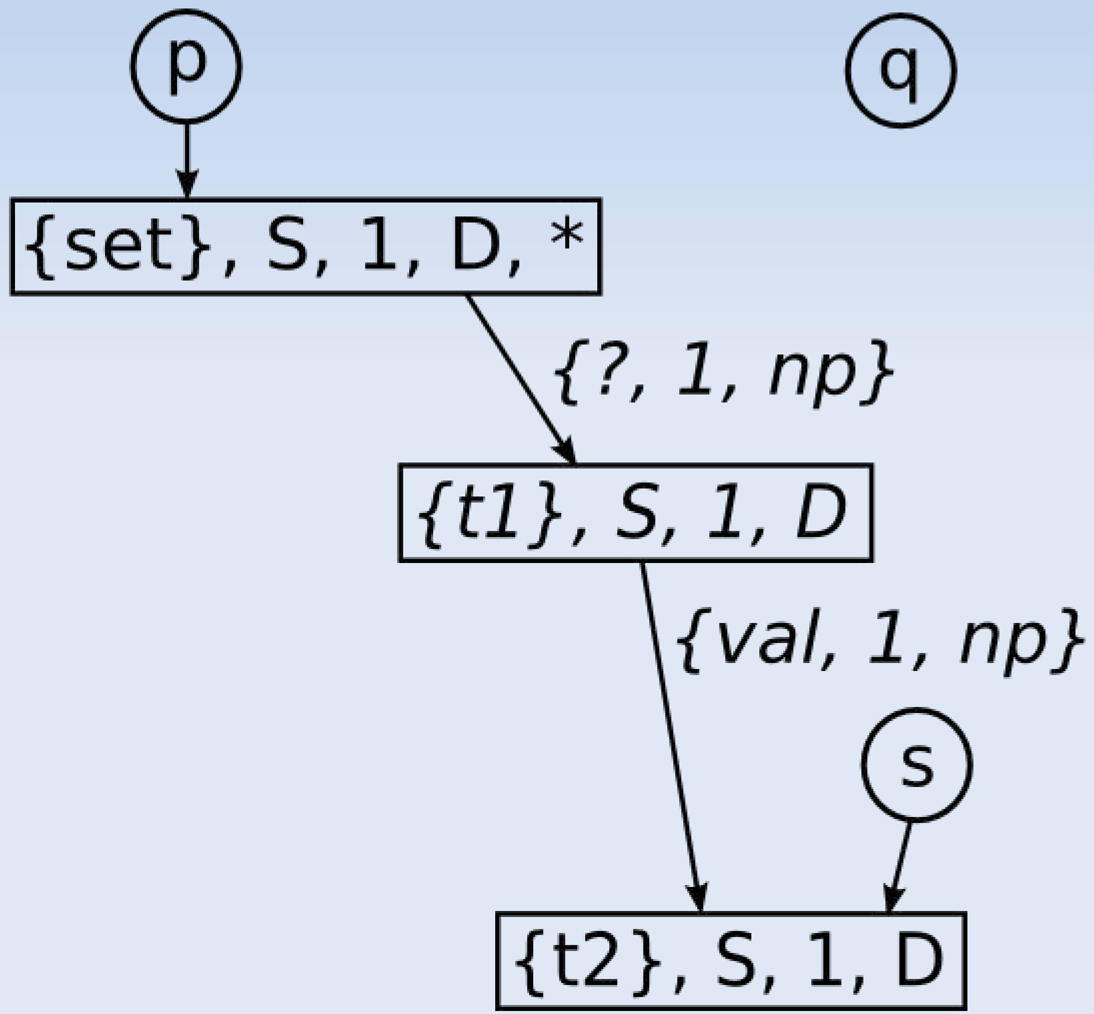


Memory locations c and d are ***connected***.
Edge E is ***interfering (ip)***.

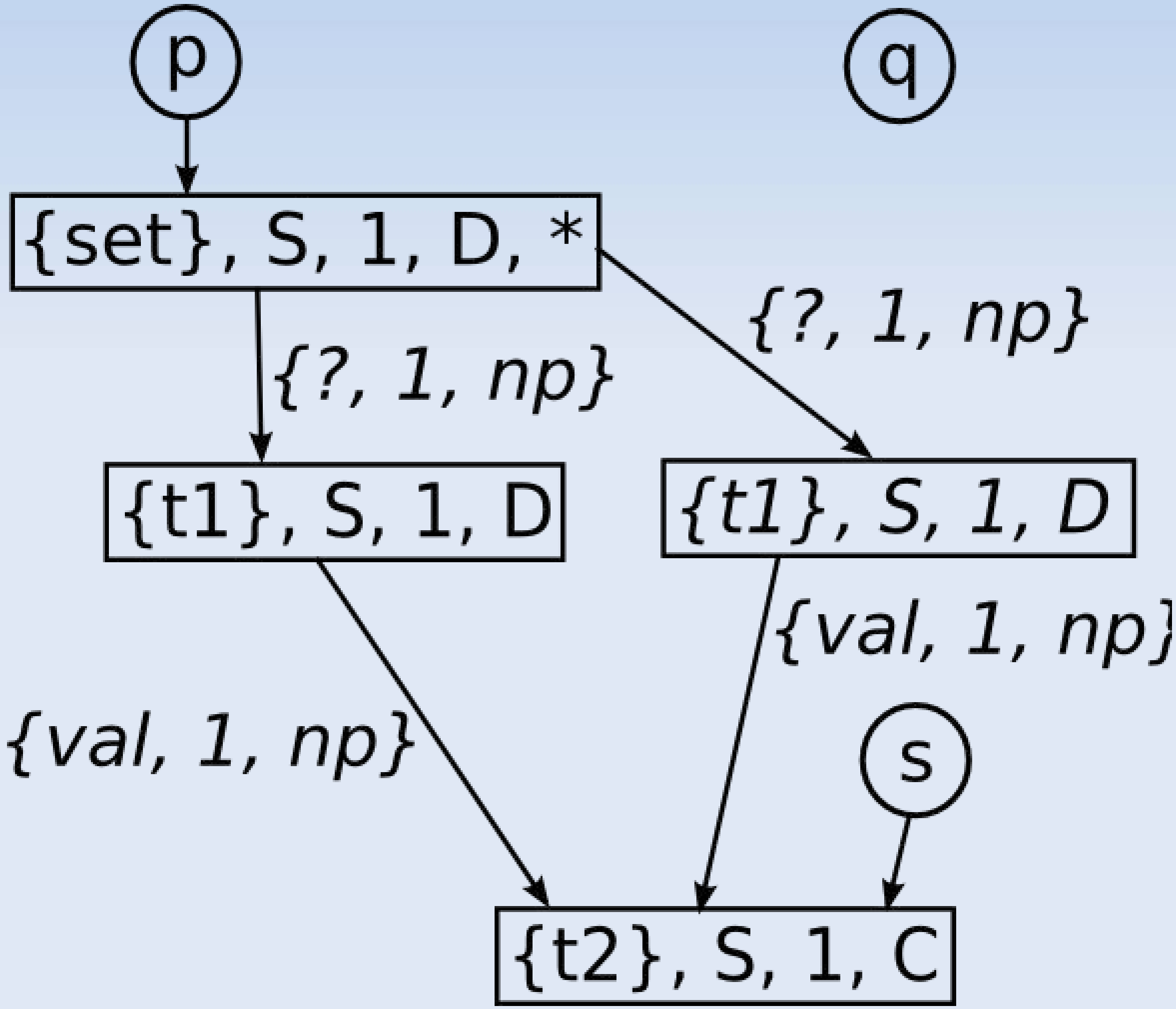
Fill a Set

```
set<t1> p = new set<t1>()
t1 q
t2 s = new t2()
for(int i = 0; i < MAX; ++i)
{
    q = new t1()
    q.val = s
    p.insert(q)
}
```

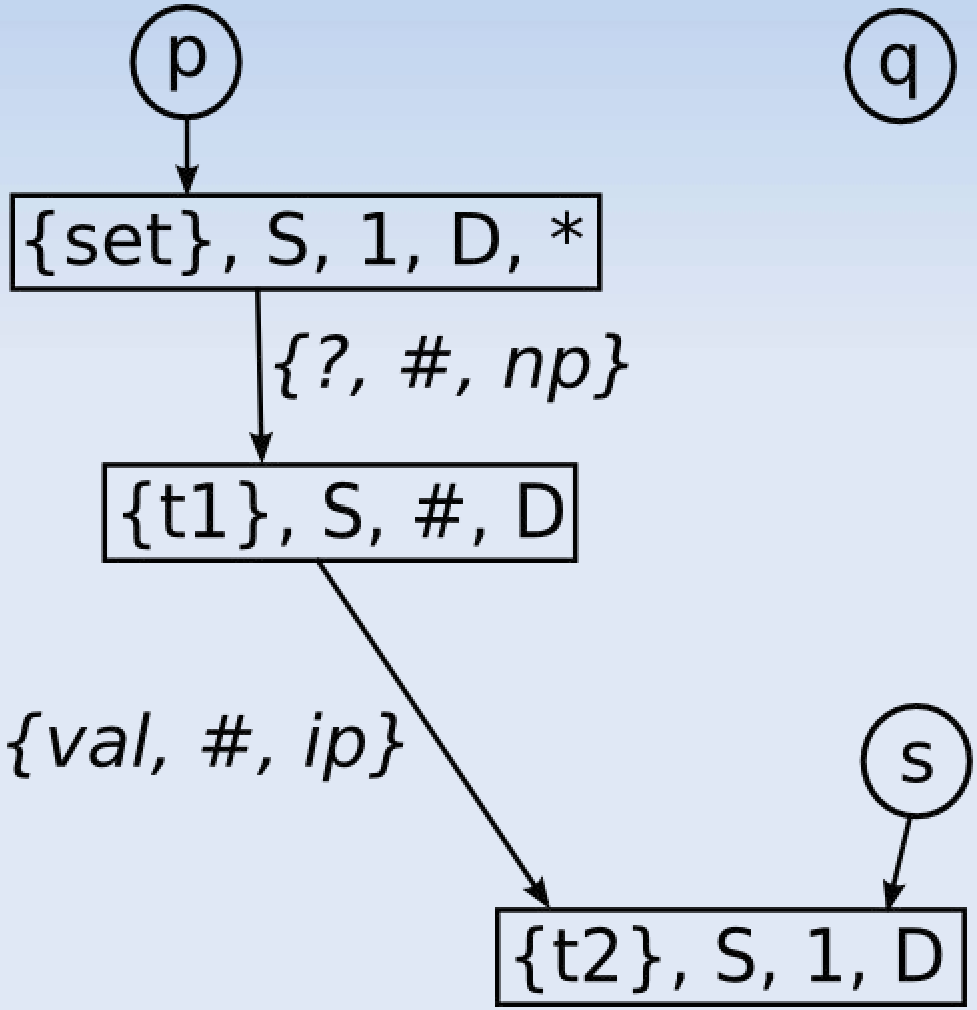
Added First Element



Added Second Element



Normal Form + Fixpoint



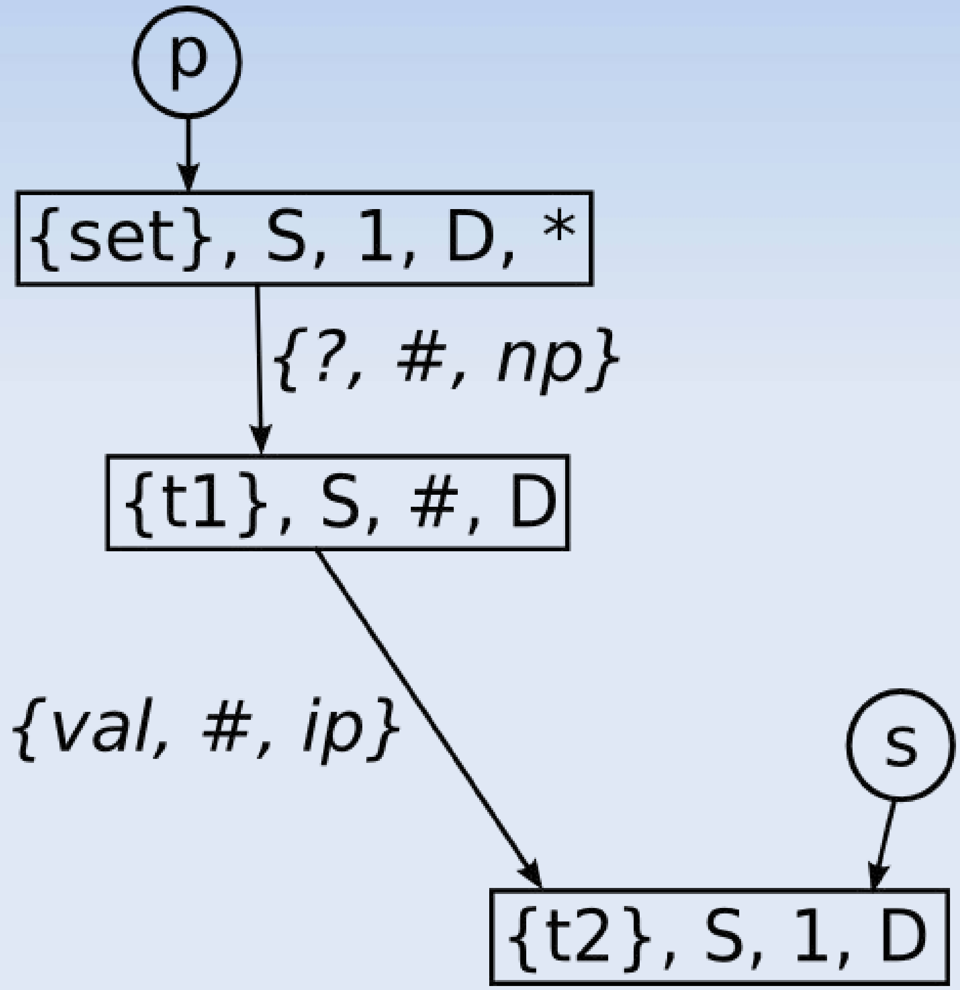
Partitioning the Pointers in a Collection

- We have seen the special identifier for the summary edge ”?”
- Now we use the order induced by the collections iterator to define 3 other identifiers
  - ”@” to represent the specific pointer in the collection of interest
  - ”B@” to represent all the pointers that come before the pointer of interest
  - ”A@” to represent all the pointers that come after the pointer of interest

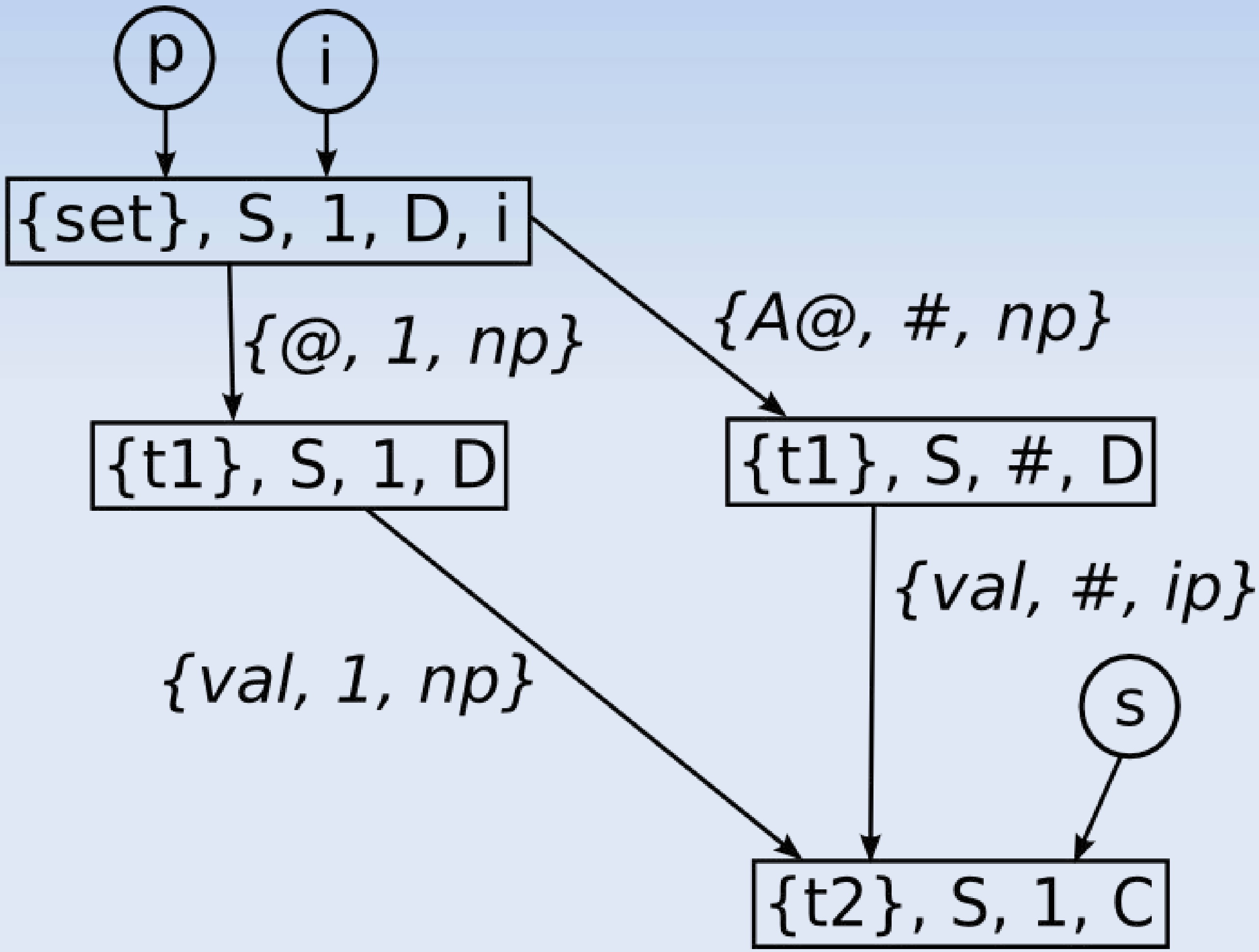
Refinement

- The ”?” edge is split into two/three new edges one representing the specific pointer of interest ”@” and the others representing the rest of the pointers ”B@” and ”A@”.
- The nodes can be split as well, into the specific targets of the ”@”, ”A@” and ”B@” edges.
- Other edges are then split as needed to connect the newly created nodes.

Initial Set



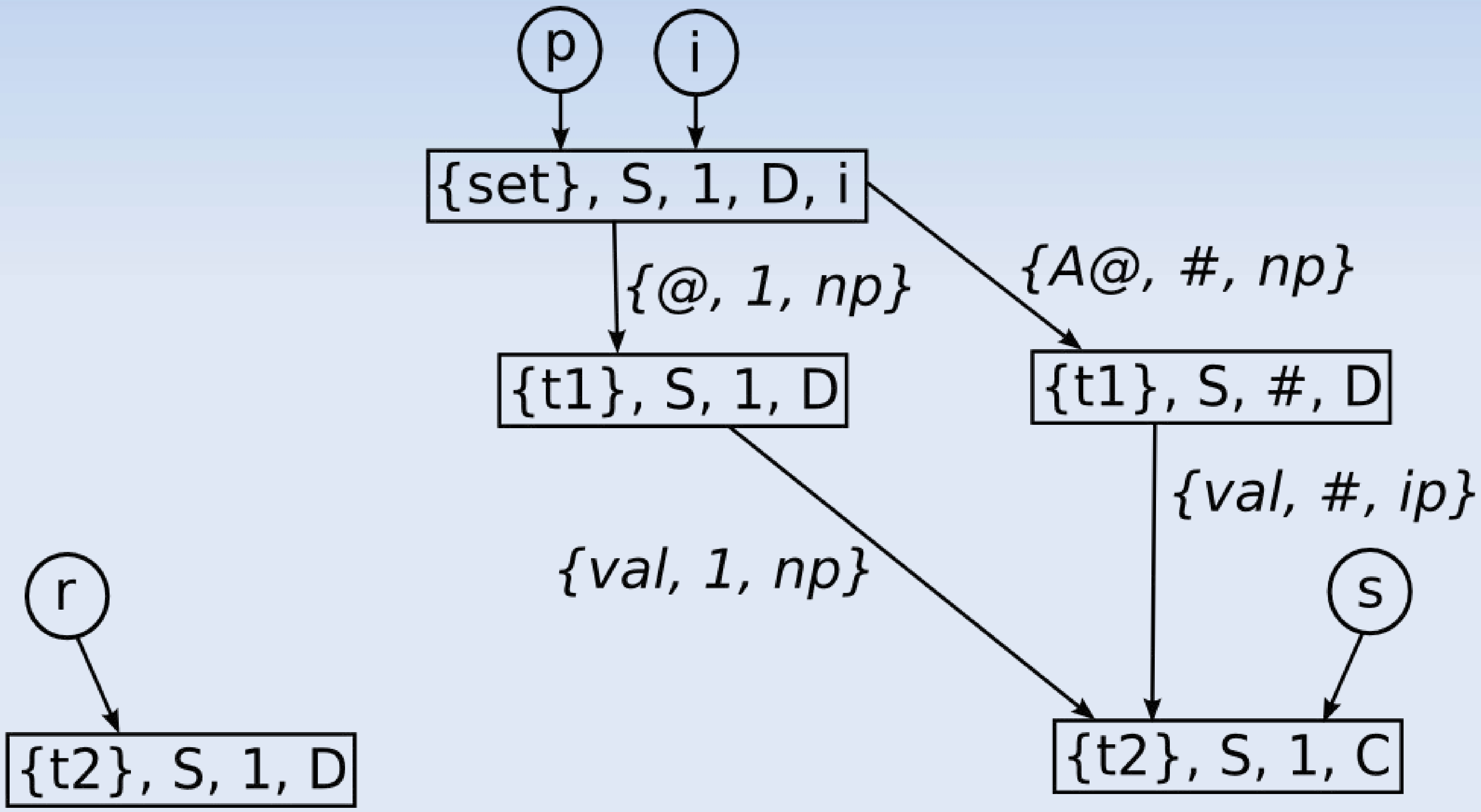
Refined Heap for Iterator Begin



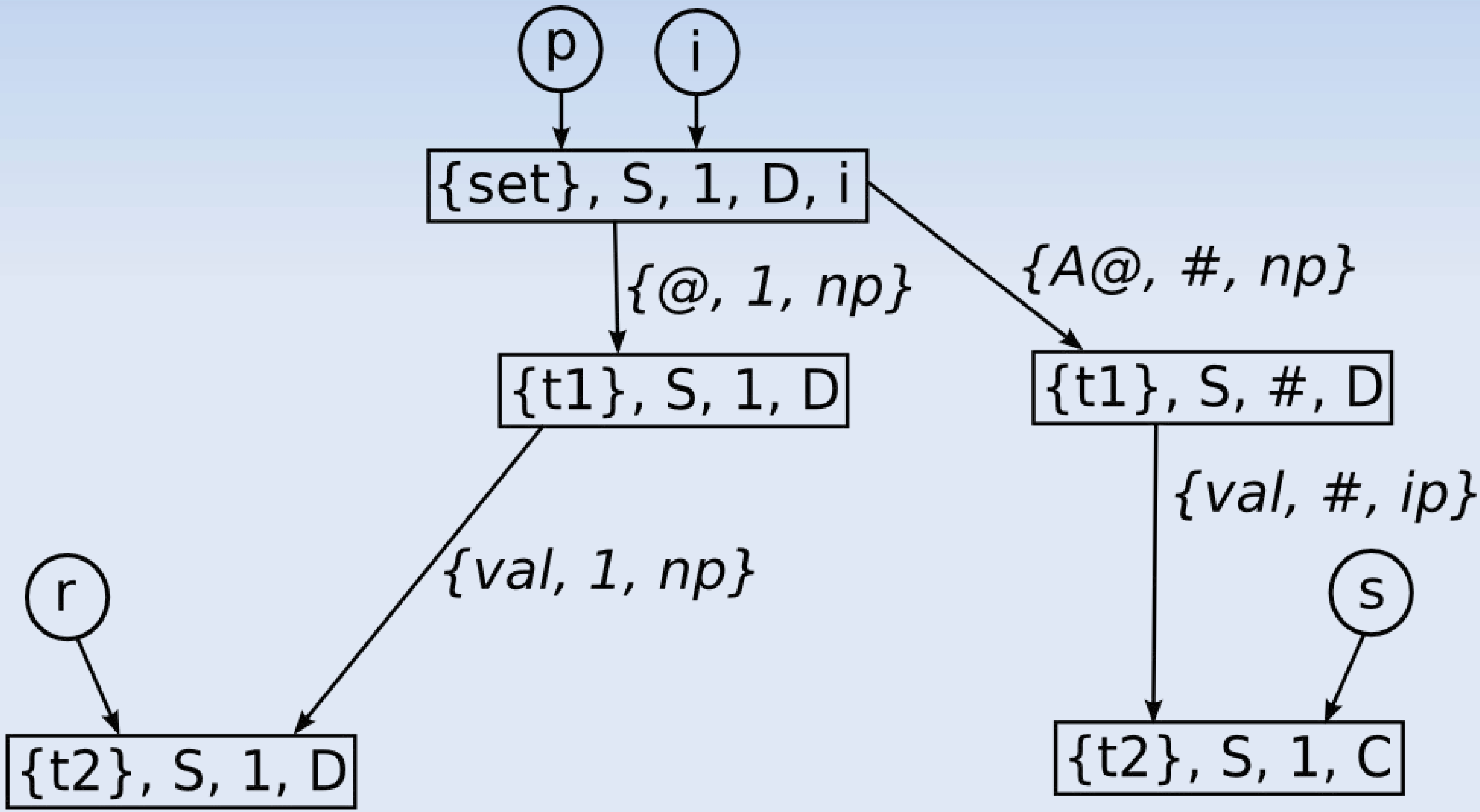
For Each in Set

```
t2 r = new t2()
iterator i = p.begin()
while(i.isValid())
{
    (i.get()).val = r
    i.advance()
}
```

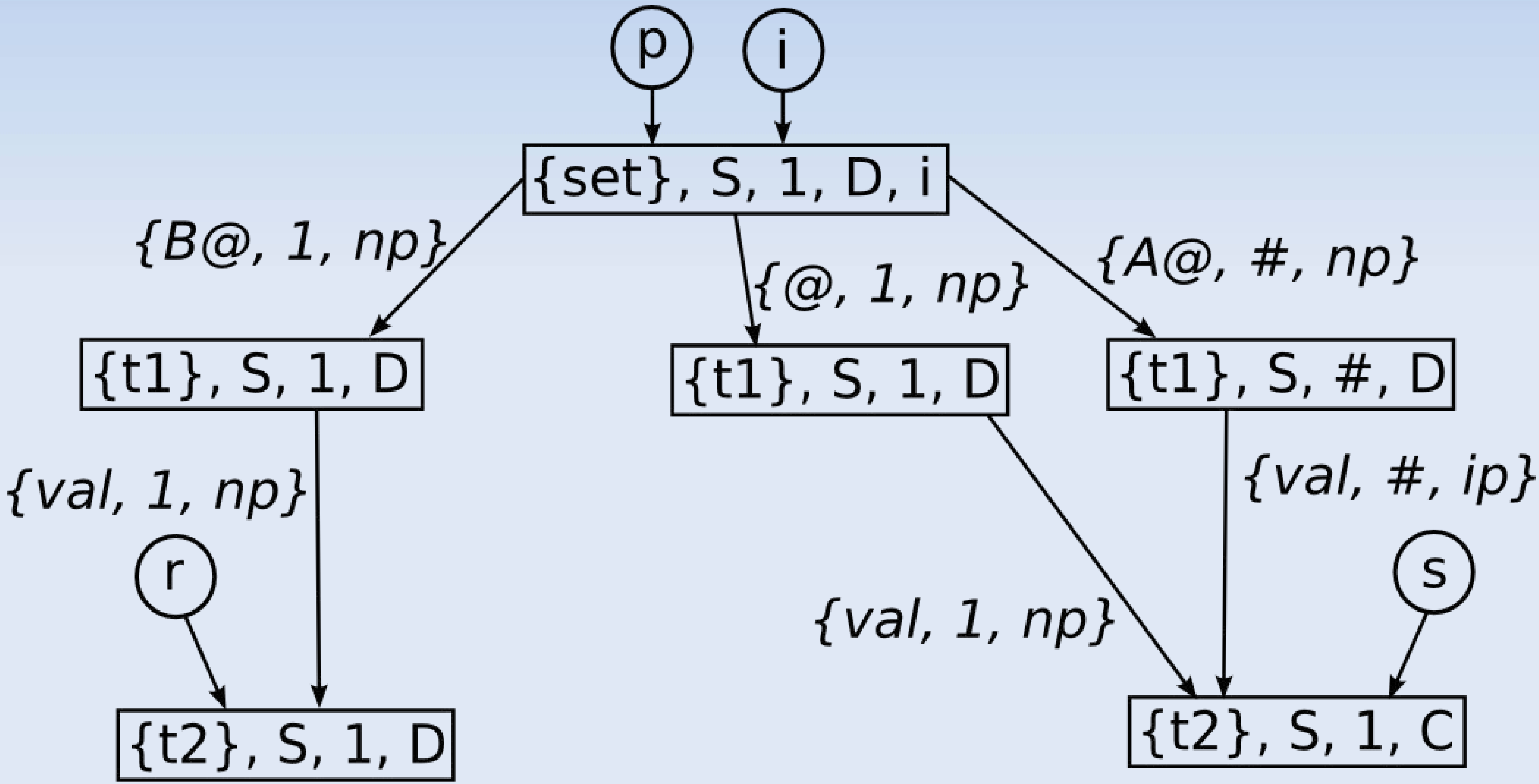
Initalize Iterator and Allocate New Target



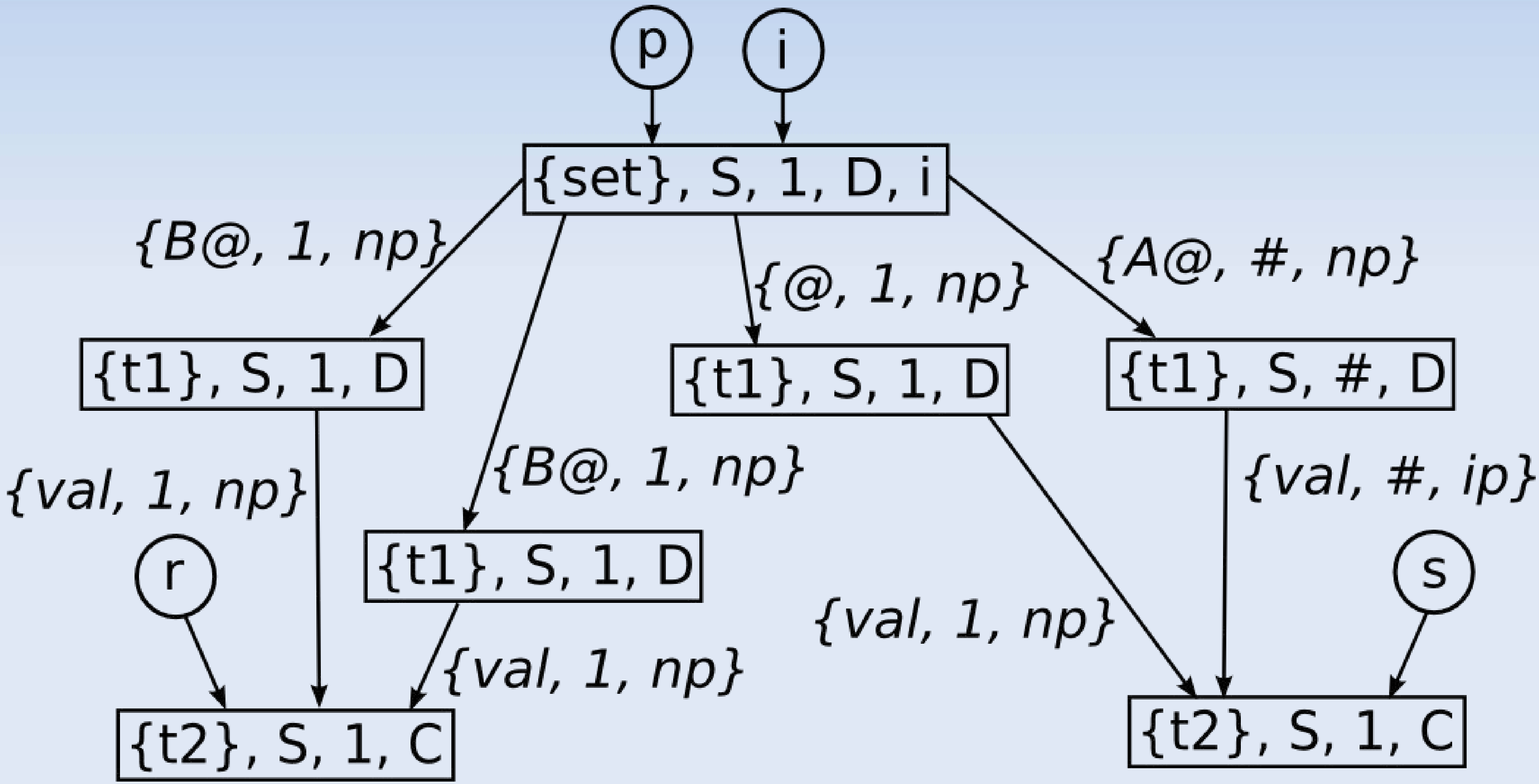
Update Current Element



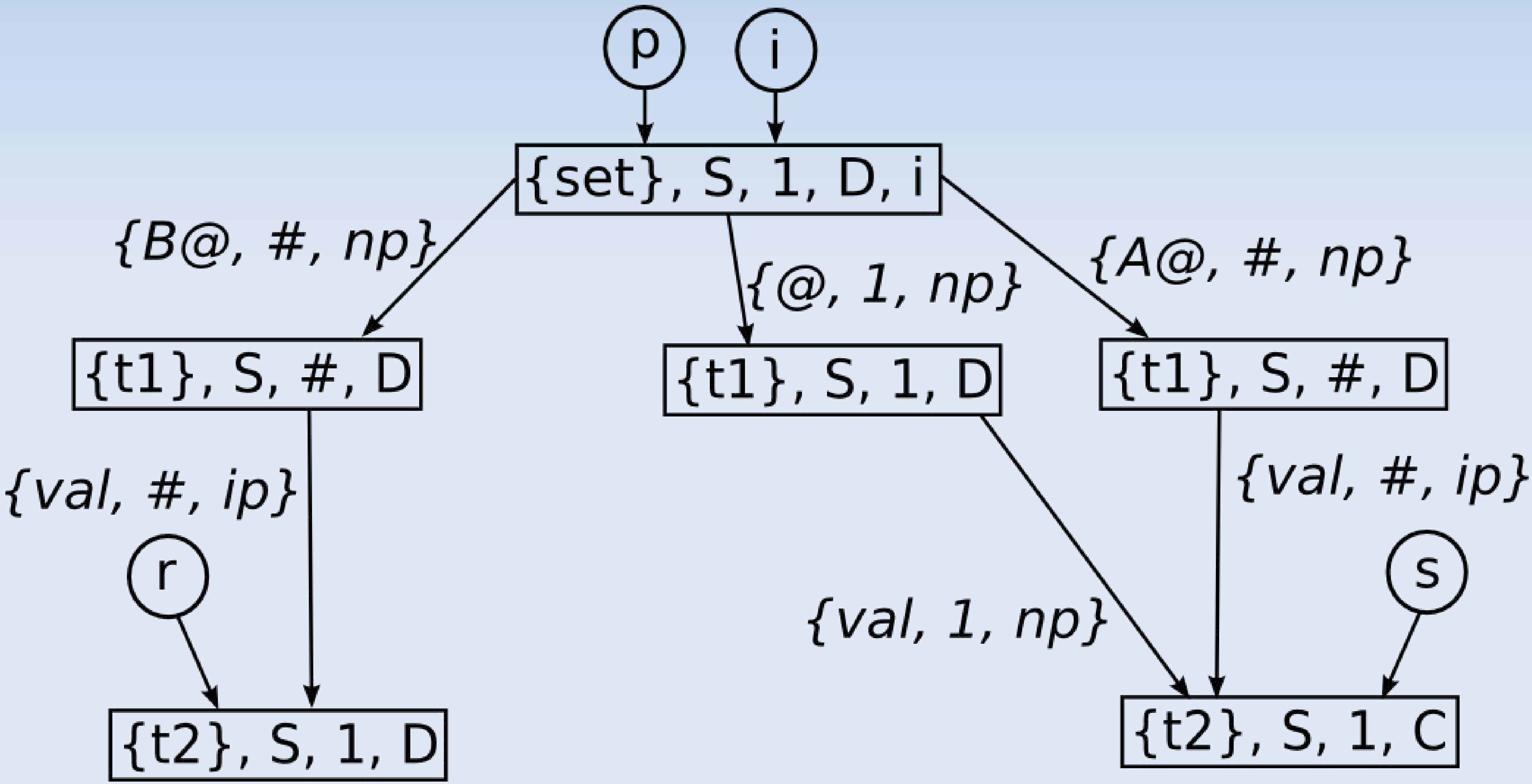
Advance Iterator



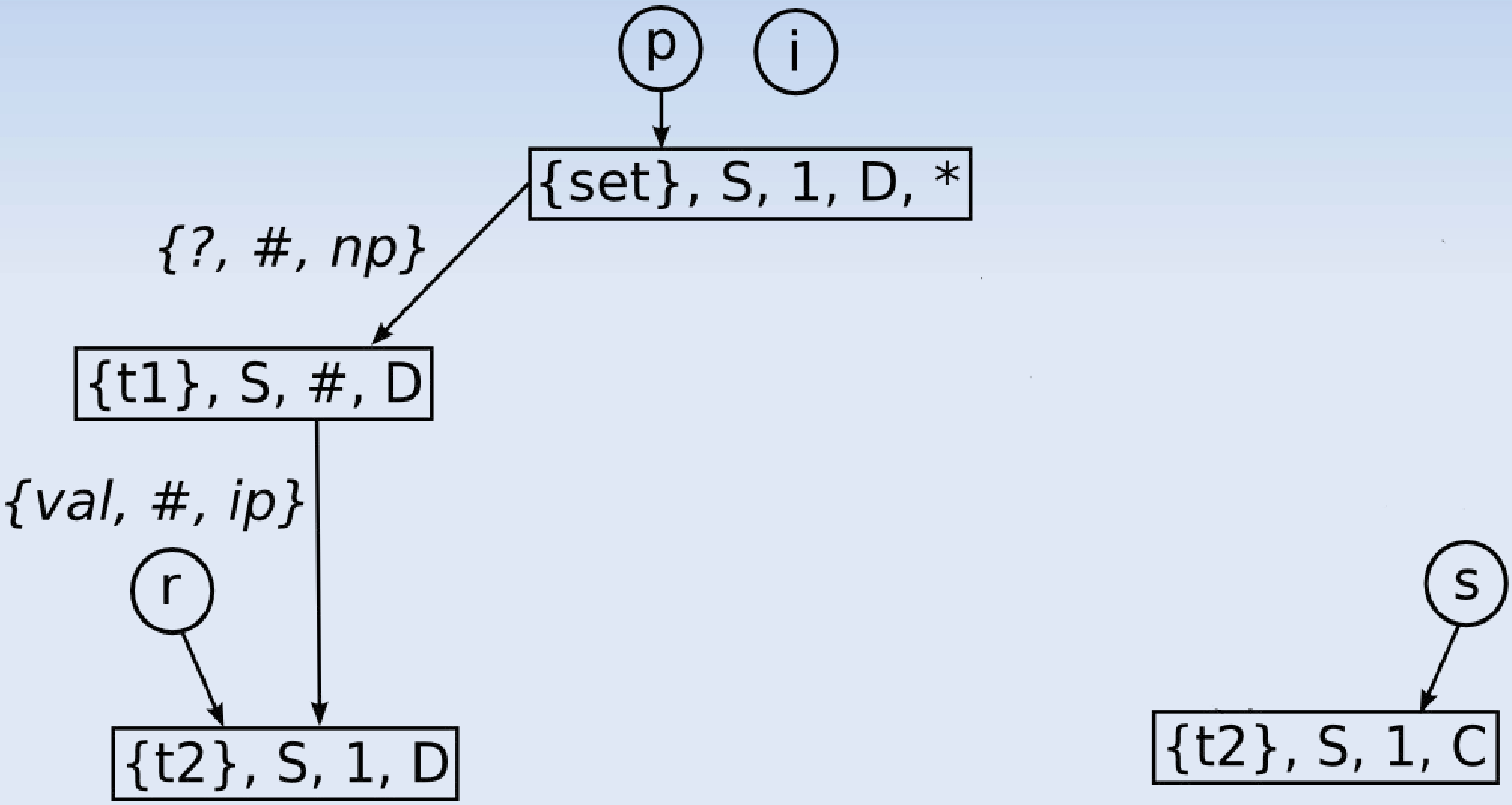
Second Assign and Advance



Normalize and Fixpoint



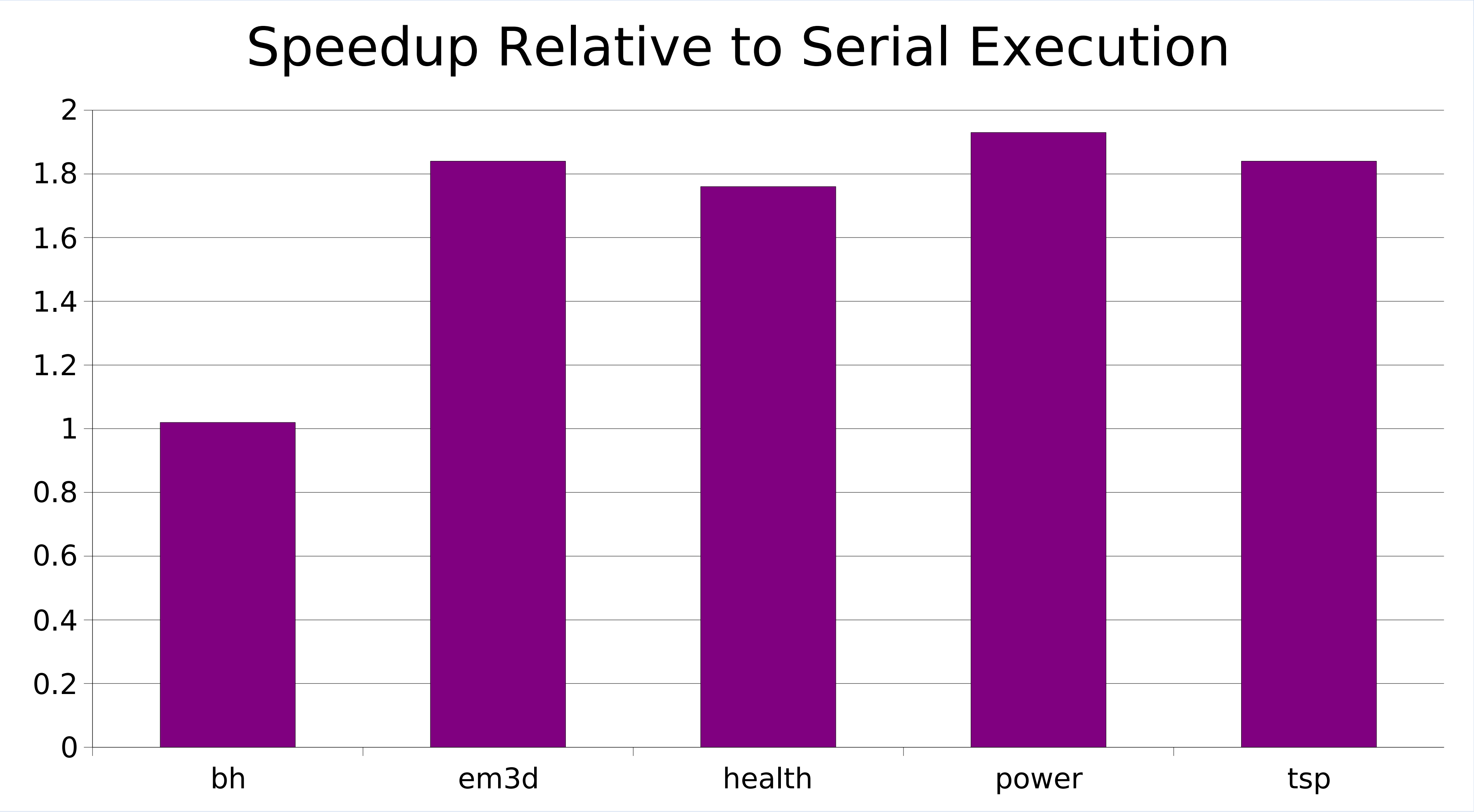
Interpret isValid as False



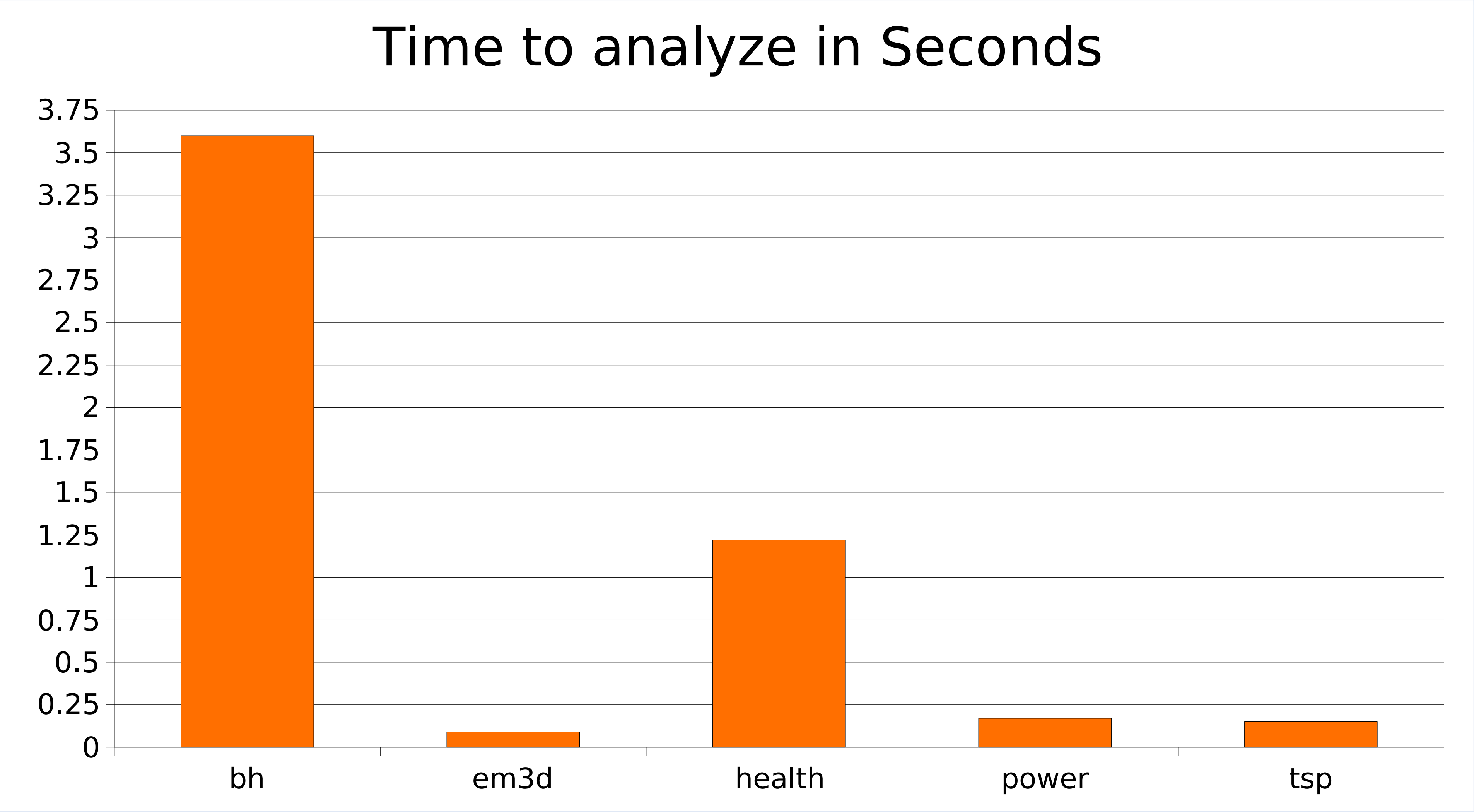
Experimental Analysis

- To test the accuracy of the shape analysis results we utilize the information to perform thread-level parallelization
- We modified a number of the Jolden benchmarks to use our collection libraries
- Based on the shape information we parallelized loops and tree calls
- Our test machine is a (dual core) Pentium D at 2.8GHz with 1GB of RAM

Parallelization Results



Analysis Runtime



Conclusion

- Our semantics-based approach to modeling collections is an effective way to handle them in a shape analysis.
- The major issue is the transformation from summary representations of the contents into more explicit forms.
- The notion of iterator order is an effective technique to model progress in the processing of collections.
- These features enable strong updates on single elements in the collection and on the collection itself.

Questions?