Krenz Security Architecture
Programatica case study

Peter D. White

Lunch is next!

Outline

- Application case studies
- The separation concept
- The Krenz concept
- Krenz and NetTop
- Building the Krenz specification
- The recursive graph concept
- The theorem proving effort
  - Hol proof of: deepen . flatten = id
- Next steps

Application case studies

Separation and Krenz

Krenz application of Programatica

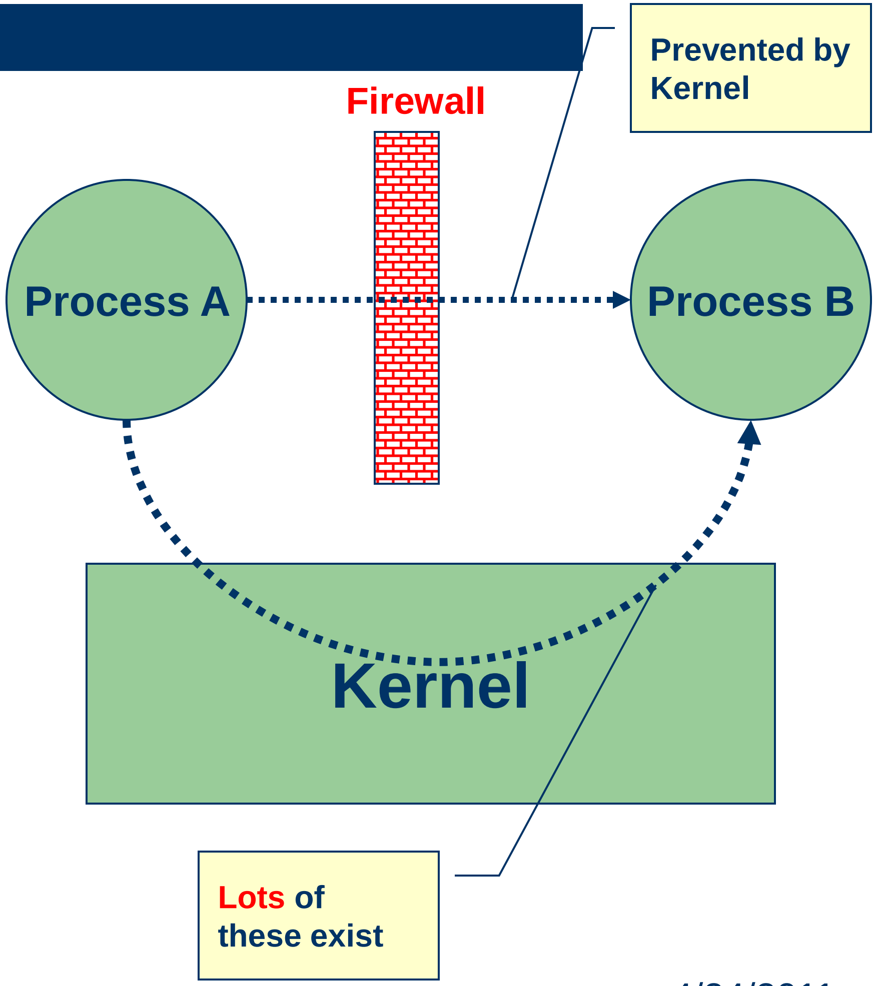
- Provide an industrial strength test case for Programatica
  - Specification and program development
  - Theorem proving
- Provide the foundation for a Krenz kernel
  - Security policy model useable both by Krenz and NetTop
  - Kernel implementation with a Posix like interface

The separation concept

Getting closer to lunch

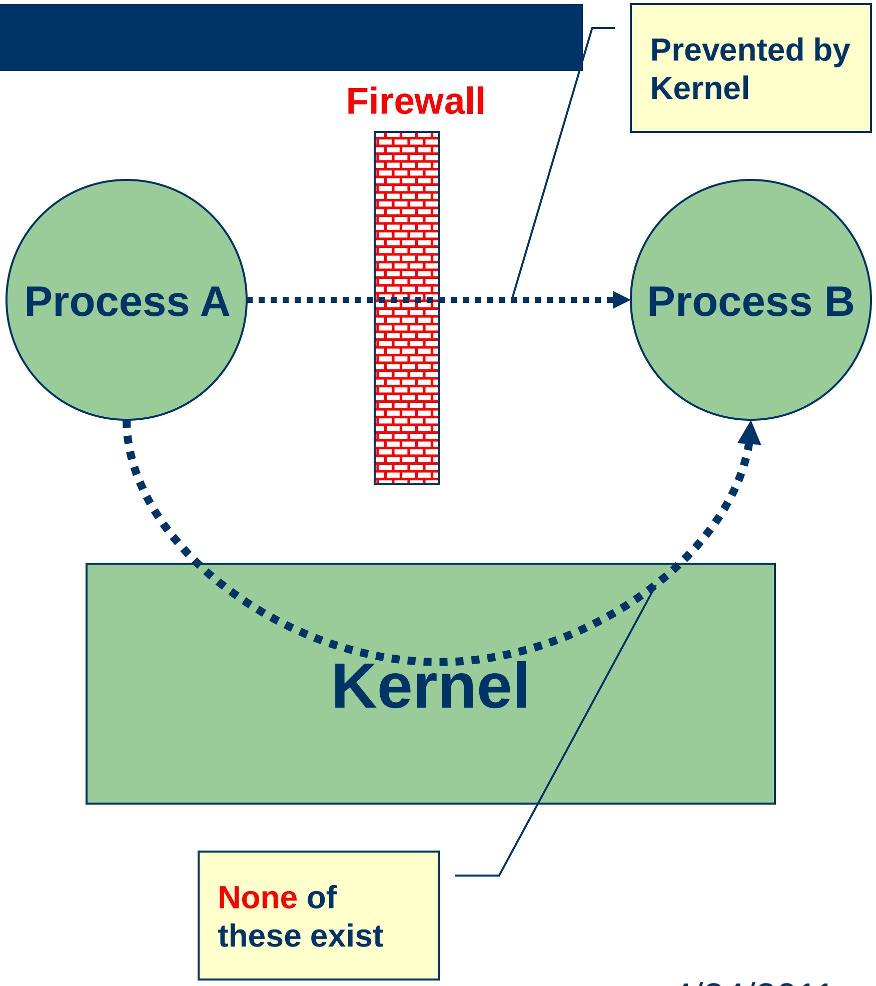
Process Protection

- Protection can be used to prevent direct interaction of processes
  - Separate logical address space
  - File system permissions
- Lots of communication pathways exist via the kernel itself
  - Resource limits
  - Resource availability
    - E.g. ports, sockets
  - Unadvertised communications paths



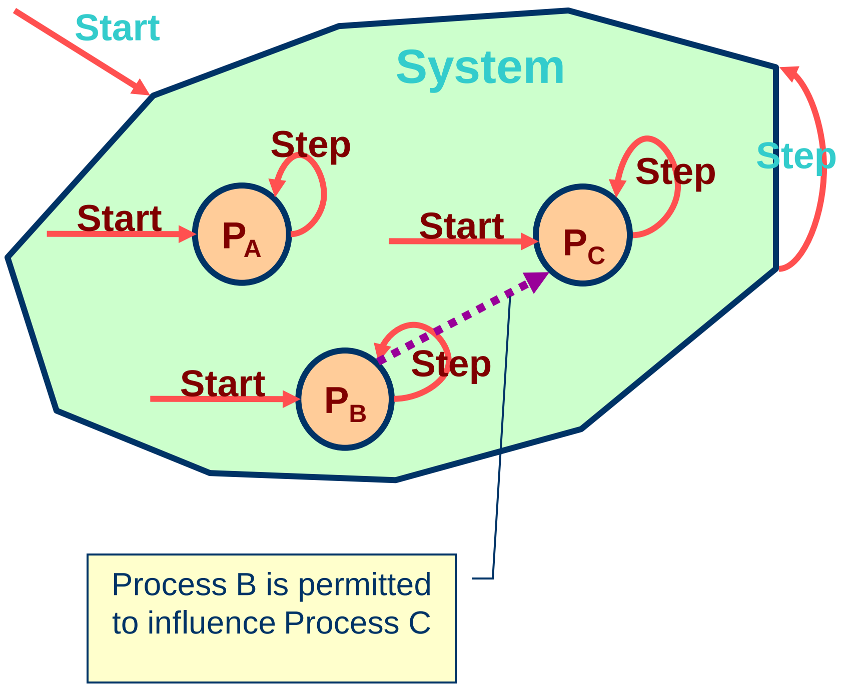
Process Separation

- Still use process protection mechanisms
  - Separate logical address space
  - File system permissions
- No communication pathways exist via the kernel itself
  - This implies a careful design of the kernel to meet the separation policy



A separation policy

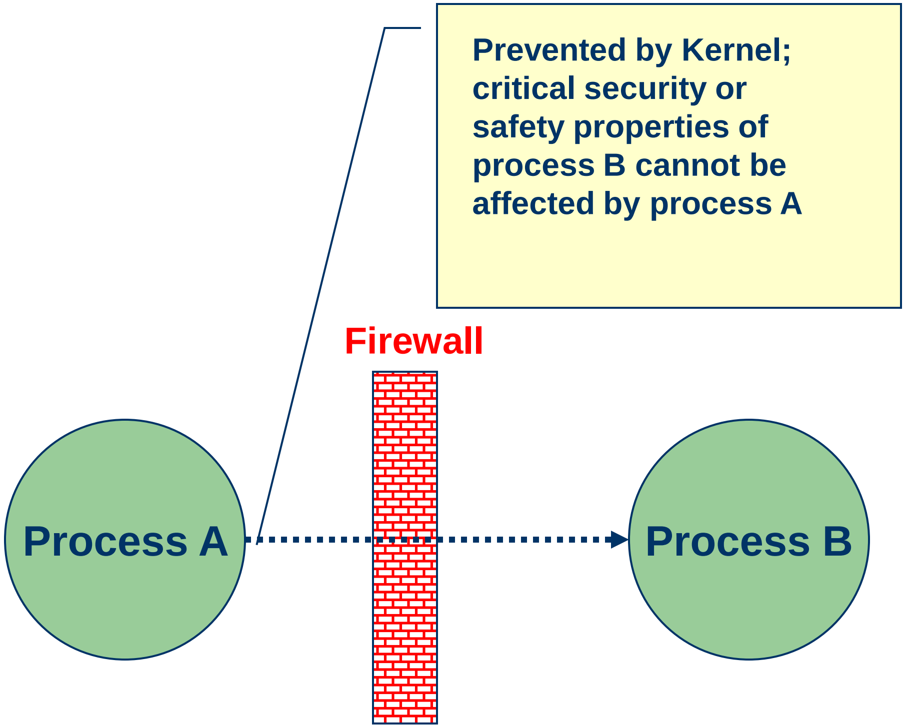
- The kernel permits interaction between processes if and only if explicitly allowed by the policy
- The policy is a directed graph of processes
  - The example here has only three nodes and one edge



Some processes are permitted to affect each other, some are not

Why a separation kernel?

- For high confidence applications
  - High assurance of red black separation
  - High assurance of fault tolerance
- In the absence of separation
  - Cause and effect tend to be local, however
  - Anything could affect anything

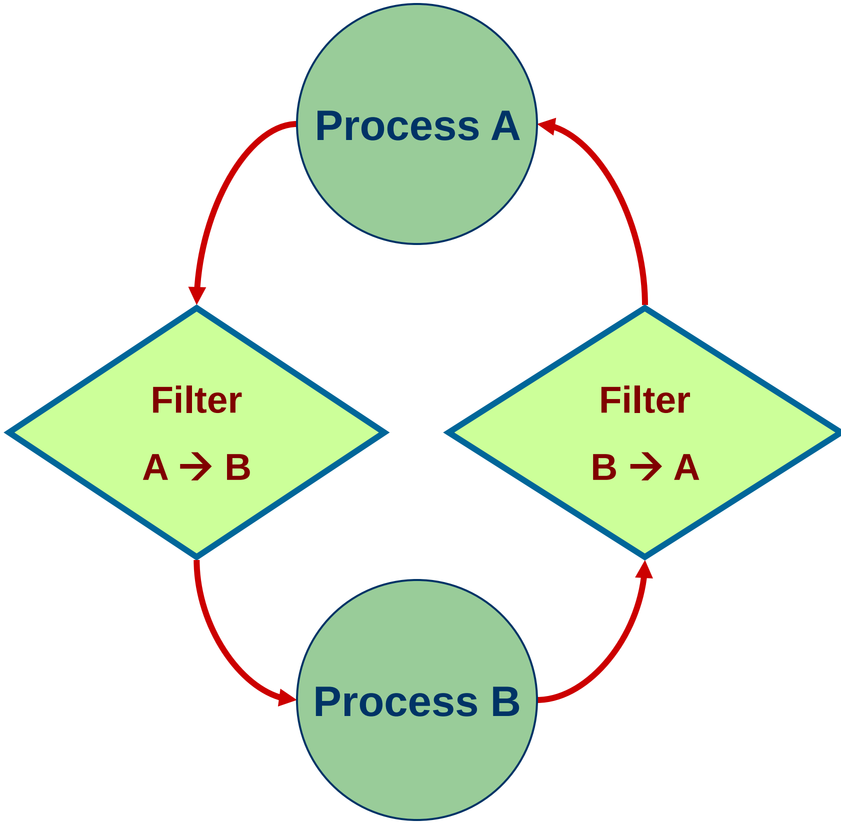


The Krenz Concept

Enhancing the separation concept

The Krenz concept

- Replace communication policy with filtered communication policy
  - Communication from A to B is permitted only if filtered by the A → B filter.
- Krenz policy is also a directed graph, with a property (filter) associated with each edge
- Krenz policy was partly a result of an industry survey to determine information security needs



Why the Krenz Concept?

- Separation policies are good at:
  - Prohibiting some flows of information completely
  - Permitting some flows of information without restriction
- Separation is the basis for establishing security and safety critical properties

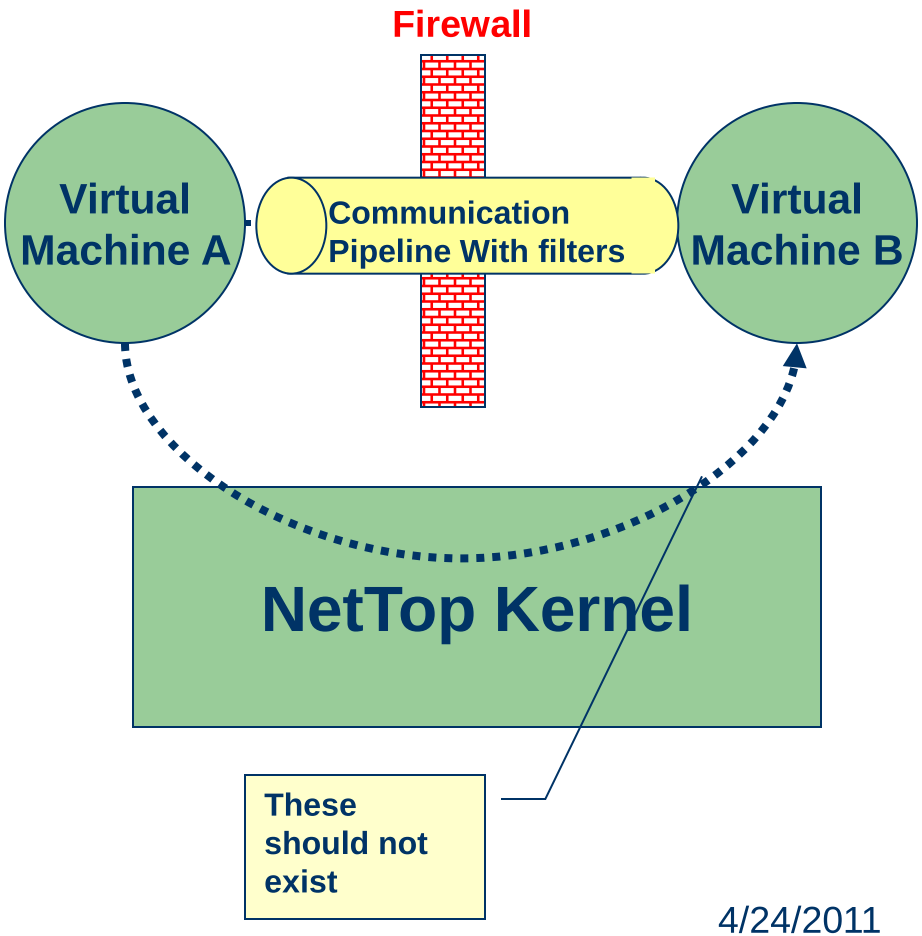
- A Krenz policy is good at:
  - Prohibiting some types of information flows (e.g. viruses)
  - Permitting information flows with restriction (e.g. encryption, signature, …)
- The Krenz policy captures naturally what most security policies are about

Krenz and NetTop

A brief diversion from the Programatica work

Krenz and NetTop

- NetTop provides separation between virtual machines hosted on Linux
- NetTop permits a communication pipeline between virtual machine when specified by policy
- The Krenz security concept is a match for what NetTop does
- Krenz can provide a model for how NetTop can be used to construct networks in accordance with a security policy
- Historical note: Krenz resulted partly from an attempt to provide cots OS and applications with a high degree of assurance



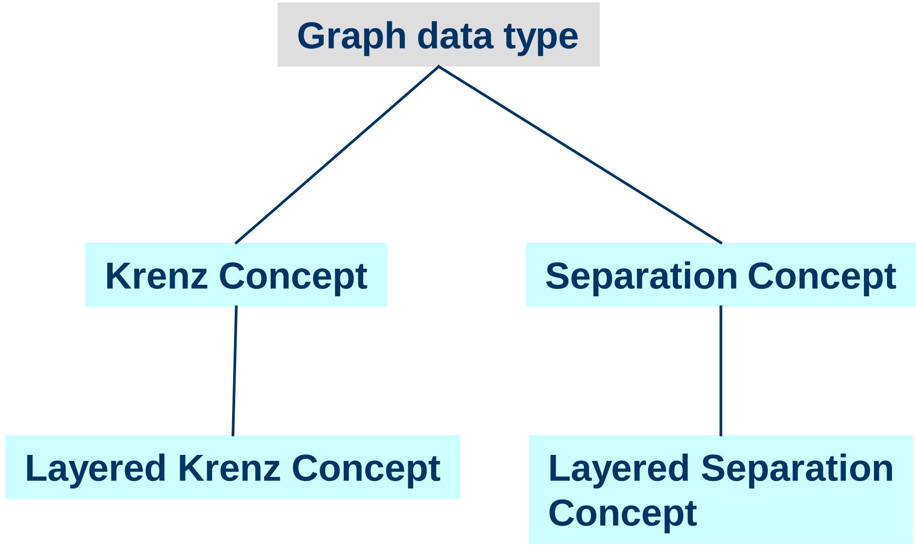
Building the Krenz specification

The recursive graph concept

Haskell construction of the Krenz

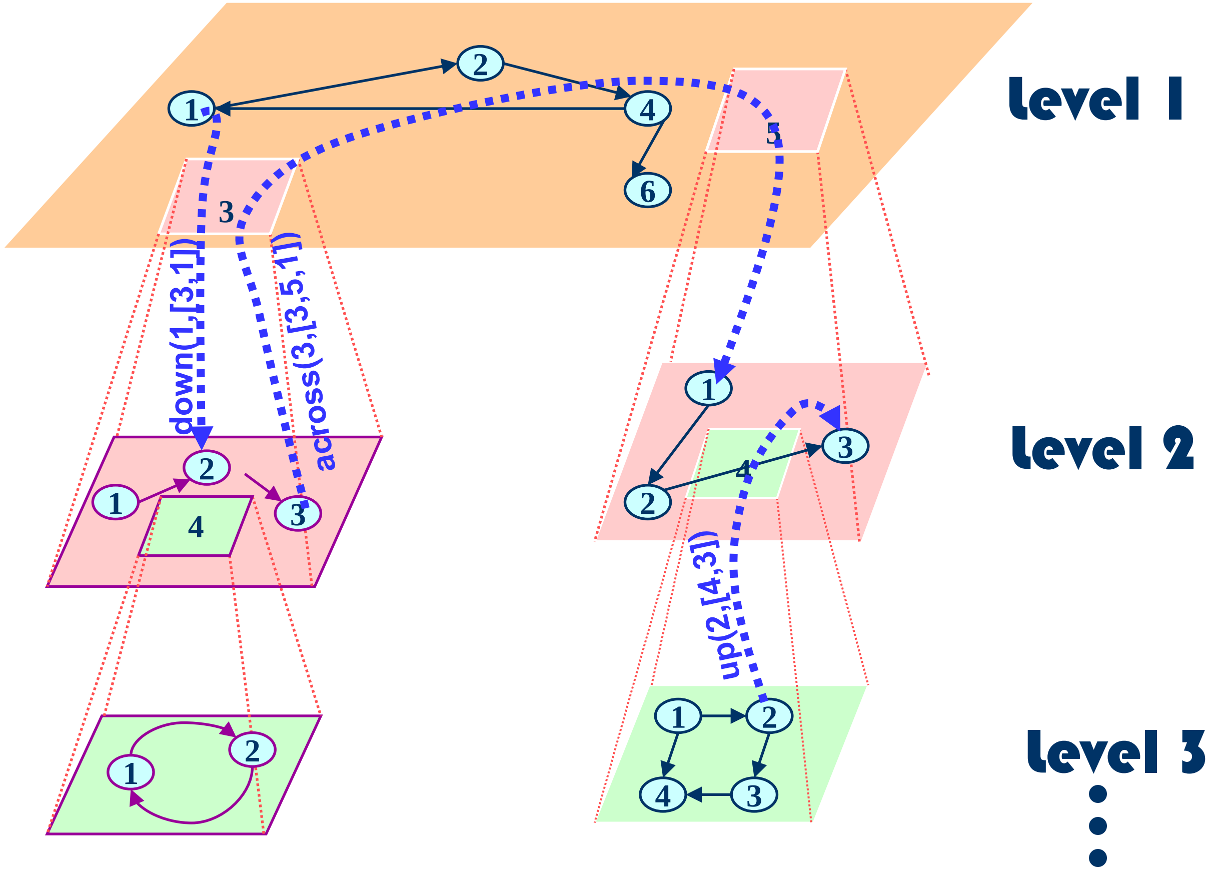
- A directed graph data type is defined, and Krenz and Separation are defined in terms of the Graph data type
- However: want to apply the Krenz concept to a graph of coalitions
  - Each coalition is a network, with its own Krenz policy
    - Each network has sub networks, with their own Krenz policy
      - Each sub network has platforms, with their own Krenz policy
        - …

Simple graph data type provide a basis for the separation and Krenz specifications



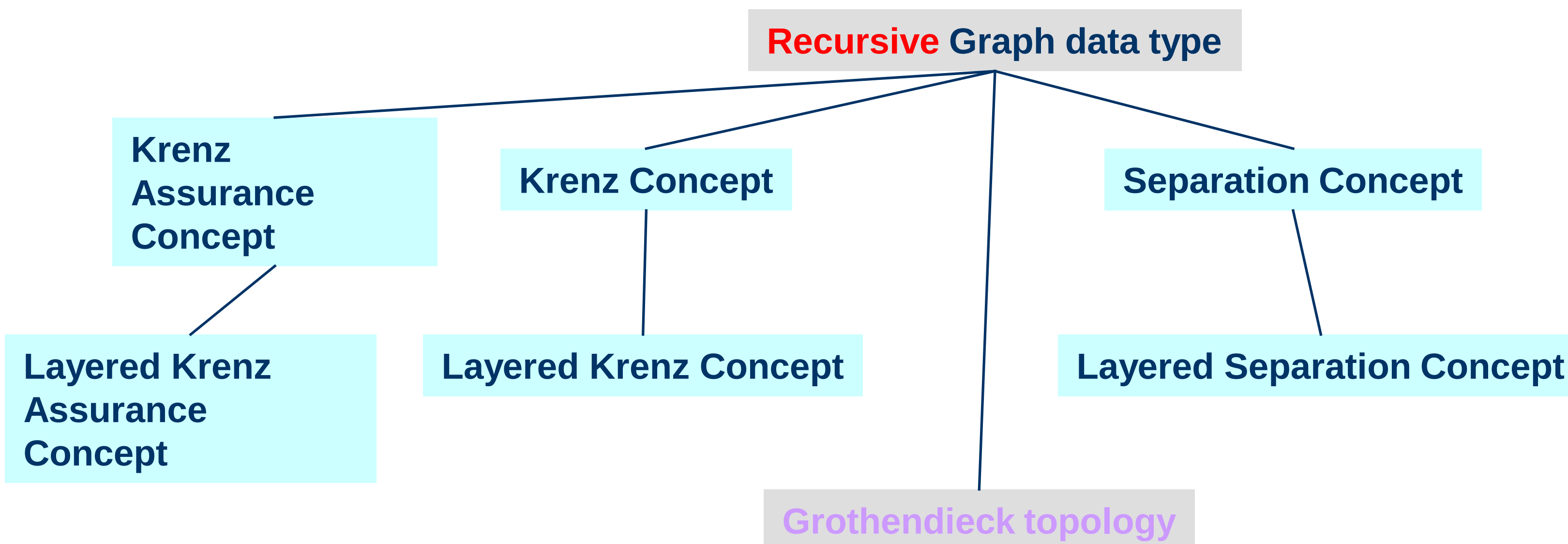
Simple graph data type inadequate for the layered Krenz

The recursive graph concept



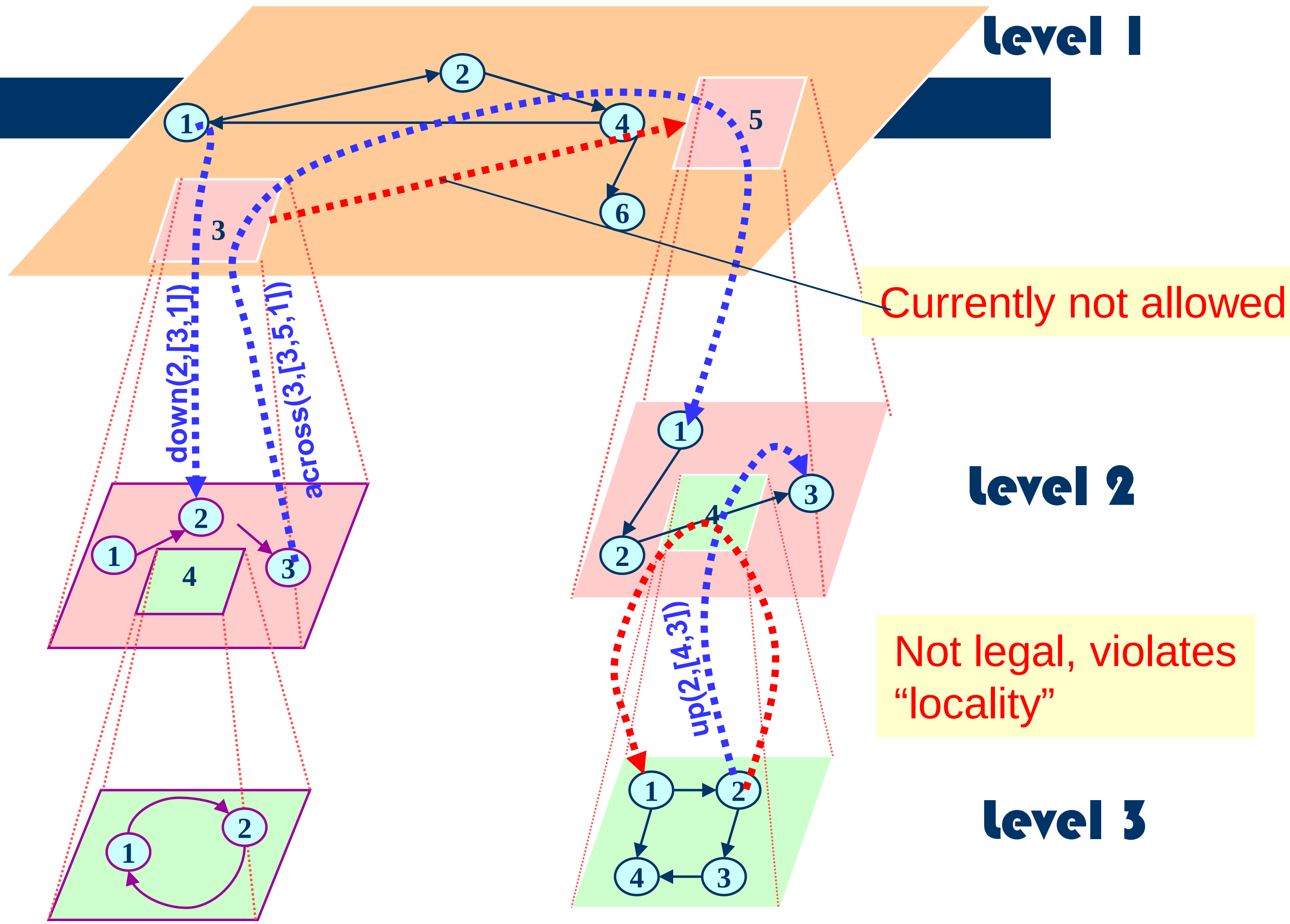
The hierarchy of data types

A lot hinges on how well recursive graphs are defined!

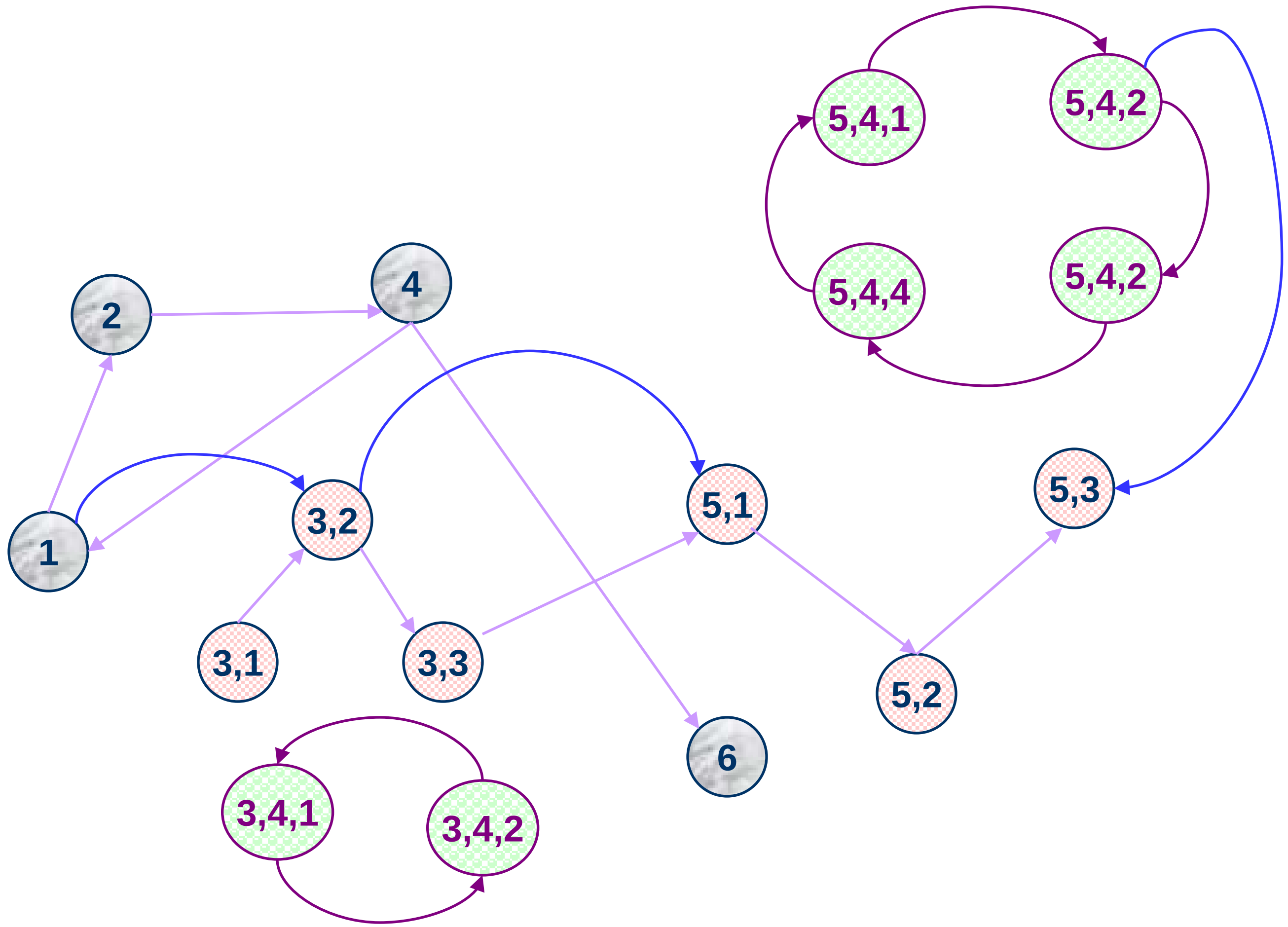


Providing the basis for recursive graph pattern matching (based on the work of Y.V. Srinivas)

Recursive Graph Structure



Flattened Recursive Graph



Programatica property

- Naïve property:
  – deepen . flatten = id
- However, the flatten function was defined with an accumulator argument, to keep track of where it is in the flattening process. A less naïve property to prove is:
  – !a. deepen . (flatten a) = id
  – Actual statement: !a g. deepen (flatten a g) = g

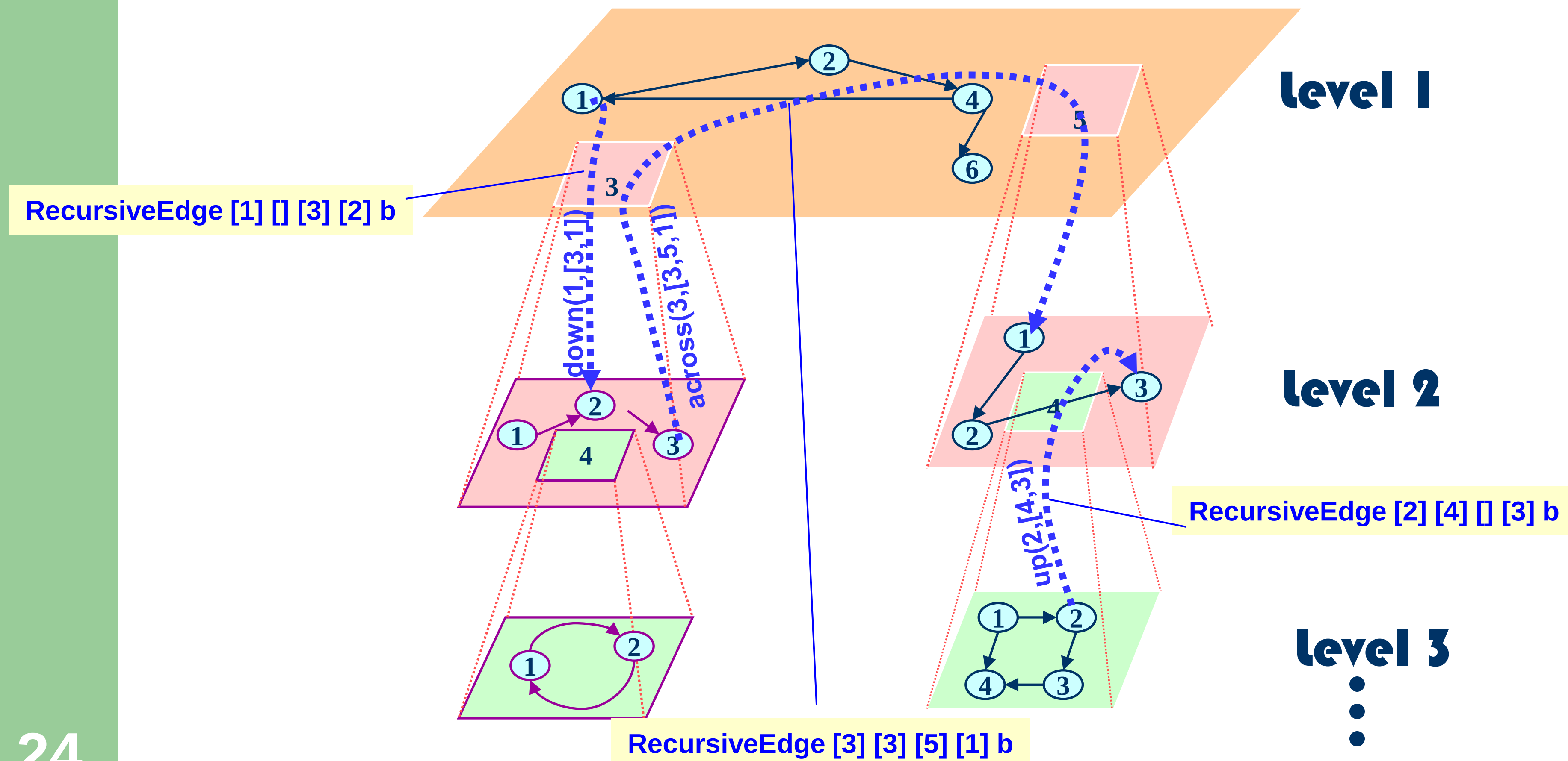
Graphs defined inductively (Martin Erwig)

- Building blocks
  - type Node a = (Int, a)
  - type Edge b= (Int, Int, b)
  - type Adj b = [(b, Node)] – Edges listed by their labels
  - type Context a b = (Adj b, Node a, Adj b, b)
  - type Decomp a b = (Mcontext a b, Graph a b)
  - type Graph a b = -- abstract type
- Constructors
  - empty :: Graph a b
  - embed :: Context a b -> Graph a b -> Graph a b
    - A graph is built inductively by adding contexts.
    - A context is a new node, with a list of predecessor and a list of successor nodes (which should already be in the graph)
- Destructors
  - match :: Node -> Graph a b -> Decomp a b

Recursive graphs defined inductively

```
– type NodeComponent = Integer
– type NodeName = [NodeComponent]
– data RecursiveNode a b =
       SimpleNode    NodeName a |
       RecursiveNode NodeName (RecursiveGraph a b) a
  data RecursiveEdge b = RecursiveEdge {
   reSource    :: NodeName,
   reUplink    :: NodeName, -- For going up in the graph
   reDownlink  :: NodeName, -- For going down in the graph
   reSink      :: NodeName,
   reEdgeLabel :: b
   }
– data RecursiveContext a b = RecursiveContext {
      preds :: [RecursiveEdge b], -- List of predecessors
      node  :: RecursiveNode a b, -- Node to add
      succs :: [RecursiveEdge b]  -- List of successors
      }
– type Decomp a b =
–          (Maybe (RecursiveContext a b), RecursiveGraph a b)
– data RecursiveGraph a b =
      EmptyRecursiveGraph |
      RecursiveGraph (RecursiveGraph a b) (RecursiveContext a b)
– See also well formed graph
```

The recursive graph concept



Hol version of Recursive Graph

```
– val x = Hol_datatype
      `RecursiveNode = SimpleNode of int list => 'a |
                       RecursiveNode of int list => RecursiveGraph => 'a;
       RecursiveEdge =
         <|source:   NodeName;
           uplink:   NodeName;
           downlink: NodeName;
           sink:     NodeName;
           edgeLabel: 'b
          |>;
       RecursiveAdjacency = RecursiveAdjacency of RecursiveEdge list;
       RecursiveContext =
         <|preds:   RecursiveEdge list;
           newnode: RecursiveNode;
           succs:   RecursiveEdge list
          |>;
       RecursiveGraph = EmptyRecursiveGraph |
                        RecursiveGraph of RecursiveGraph => RecursiveContext;
       Decomp =
         <|flag: bool;
           component: RecursiveContext;
           subgraph:  RecursiveGraph
          |>`
```

<u>Here be monsters:</u> The type of RecursiveEdge is ` ``:NodeName -> NodeName -> NodeName -> NodeName -> 'a -> ('b, 'a) RecursiveEdge`` ` Note that there are two parameters ('b, 'a) according to the output, but there is really only one parameter in the definition.

The theorem proving effort

Hol proof of: deepen . flatten = id

Thinking about inductive proof

- The flatten and deepen functions follow the recursive structure of the graph itself
- This structure carries all the way down through recursive context, recursive edge, and recursive node (backup slides)

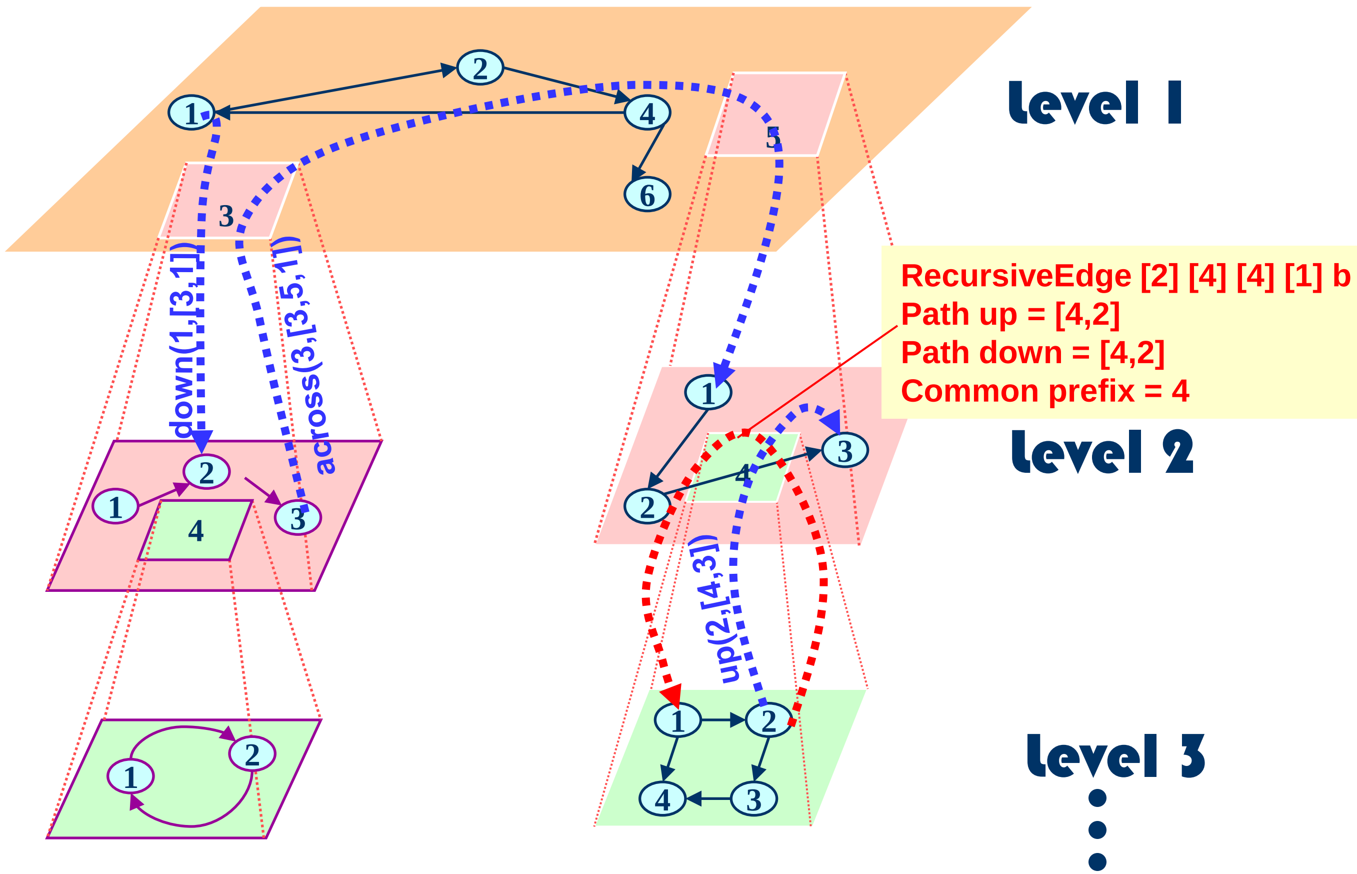
```
flatten :: NodeName ->           -- Node name at next higher level
           RecursiveGraph a b -> -- Graph to flatten
           RecursiveGraph a b    -- Flattened graph
flatten _context EmptyRecursiveGraph = EmptyRecursiveGraph
flatten context (RecursiveGraph g rc) =
       RecursiveGraph (flatten context g) (flattenContext context rc)

-- Deepen a flattened graph, restoring its recursive structure.
deepen :: (Show a, Show b) =>
          RecursiveGraph a b -> -- Graph to deepen
          RecursiveGraph a b    -- Resulting deepened graph
deepen EmptyRecursiveGraph = EmptyRecursiveGraph
deepen (RecursiveGraph g rc) = RecursiveGraph (deepen g) (deepenContext rc)
```

Well formed recursive graph

- Did not explicitly think about this until time to prove theorems
- The theorem to be proved is true only for well formed graphs
- Well formed node
  - Length of nodename is 1
  - Subgraph is well formed
- Well formed edge
  - Source and sink lengths are 1
  - Up ++ src, down ++ snk have no common prefix
- Well formed context
  - Predecessor edges, Successor edges, and node are all well formed
- Well formed graph
  - Graph and context are well formed

Well formed recursive graph



Theorem proving summary

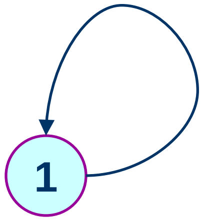
- Hol automatically adjusted formulas with overlapping patterns
  - Haskell:
    - last [h] = h
    - last (h : t) = last t – overlaps on lists of length 1
  - Hol:
    - last [h] = h (* length one list *)
    - last (h ::v2:: v3) = last (v2 :: v3) (* length >= 2 *)
  - Caused some rethinking of the proofs

Theorem proving summary

- Recursive edge problem
  - (RecursiveEdge src up dn snk b) given type with two type variables `a and `b. The type is too general
- “Ill formed induction” on graphs
  - Had to create my own subgraph relation, prove it is well founded, and construct an induction theorem
  - Later, discovered `TypeBase.induction_of (valOf (TypeBase.read "RecursiveGraph"));`

Errors found

- The common prefix of two node names of length one should be null, even when the two nodenames are the same.
  - Common prefixes can be eliminated
- The node name in a recursive node was not being addressed



RecursiveEdge [1] [] [] [1] b
Path up: [] ++ [1] = [1]
Path down: [] ++ [1] = [1]
Common prefix: [1]
Oops!

Next steps

A secure Posix or Linux like separation kernel

Problems with Linux assurance

- Monolithic kernel: A large amount of code running in kernel mode, all of which can corrupt the kernel
- Configurable and dynamic device drivers
- Some interfaces provide “covert” information flows
- Process fork: Result in child process that is clone of parent

Haskell solutions: Kernel architecture

- Construct a Kernel with device drivers that are threads in the ST monad (State monad)
  - Device driver state is guaranteed not corruptible by other kernel threads
  - Device driver can run concurrently with other kernel threads
- Linux provides standard interfaces to device drivers
  - Formulate this standard interface as a type, then the device driver is guaranteed to be a function only of:
    - The interface provided by the kernel
    - Its own state
    - Its input from the kernel

Haskell solutions: Kernel architecture

- Modular kernel
  - Provide virtual file system as separate module
    - Formulate types to ensure that the virtual file system cannot corrupt other part of the kernel, and cannot be corrupted by other parts of the kernel
  - Provide kernel IO as separate module
    - ibid
- Lazy IO
  - Many sophisticated kernel features are instances of lazy evaluation:
    - File system page with dirty bit
    - Demand paging
    - Copy on write for process cloning

Solutions: Posix API and separation

- Provide additional checks to those in standard Posix, to increase separation between processes
- Provide a comprehensive list of covert channels, by showing that the operation of a process is a function only of:
  - Its own state
  - Its input
  - A list of functions of the kernel state
    - (e.g. disk full, socket usage, …)

Summary

Summary

- Program development:
  - Design with properties is a powerful technique
- Theorem proving:
  - Want a simple minded embedding into the theorem prover
  - Test before proof: It is easier to prove the correctness of correct code
  - Make a Hol theories for the Haskell prelude, and many Haskell libraries, with lots of pre proven theorems.
  - A Haskell to Hol translator would have avoided the recursive edge problem

Summary

- Krenz development
  - Krenz provides a good model for secure systems
  - We will build a prototype Posix / Linux like kernel
    - Subset of Posix interfaces
    - Only a few higher level device drivers
    - Kernel architecture providing dynamic loading of kernel modules, with type safety providing assurance that the new modules do not corrupt the kernel

Backup slides

Thinking about inductive proof

- The structure of flatten and deepen carry all the way down

```
deepenContext :: (Show a, Show b) =>
                 RecursiveContext a b -> RecursiveContext a b
deepenContext (RecursiveContext preds (SimpleNode nn a) succs) =
    RecursiveContext
      (deepenEdges preds)
      (SimpleNode (deepenNodeName nn) a)
      (deepenEdges succs)
deepenContext (RecursiveContext preds (RecursiveNode nn sub a) succs) =
    RecursiveContext
      (deepenEdges preds)
      (RecursiveNode nn (deepen sub) a)
      (deepenEdges succs)

flattenContext :: NodeName -> RecursiveContext a b -> RecursiveContext a b
flattenContext context (RecursiveContext preds (SimpleNode nn a) succs) =
    RecursiveContext
      (flattenEdges context preds)
      (SimpleNode (flattenNodeName context nn) a)
      (flattenEdges context succs)
flattenContext context
               (RecursiveContext preds (RecursiveNode nn sub a) succs) =
    RecursiveContext
      (flattenEdges context preds)
      (RecursiveNode nn (flatten (context ++ nn) sub) a)
      (flattenEdges context succs)
```

<u>A mistake:</u> Should have made a “flattenNode” function for this, to conceal details during proof of flattenContext.

Thinking about inductive proof

- The structure of flatten and deepen carry all the way down

```
deepenEdge :: RecursiveEdge b -> RecursiveEdge b
deepenEdge (RecursiveEdge source _up _down sink b) =
    let cp = commonPrefix source sink
    in RecursiveEdge
         (deepenNodeName source)          -- Deep source is one long
         (init (drop (length cp) source)) -- Deep up
         (init (drop (length cp) sink))   -- Deep down
         (deepenNodeName sink)            -- Deep sink is one long
         b

deepenEdges :: [RecursiveEdge b] -> [RecursiveEdge b]
deepenEdges edges = map deepenEdge edges

flattenEdge :: NodeName -> RecursiveEdge b -> RecursiveEdge b
flattenEdge context (RecursiveEdge source up down sink b) =
    RecursiveEdge
      (flattenNodeName (context ++ up) source)
      []
      []
      (flattenNodeName (context ++ down) sink)
      b
flattenEdges :: NodeName -> [RecursiveEdge b] -> [RecursiveEdge b]
flattenEdges context edges = map (flattenEdge context) edges
```

Node name components are added to the end as the graph gets deeper. The proofs might have been simpler if they were added at the beginning

Embedding in Hol, a simple minded approach

- Straight translation to Hol works because I did not use existential types, monads, …

```
(* Need some Haskell stuff (we take it for granted) *)
val init_def = Define `(init [x] = [])
                 /\   (init (h::t) = h :: (init t))`;
(* Observe the "h::v2::v3 in the Hol output, this is caused by
   the pattern overlap from the Haskell prelude *)
val last_def = Define `(last [x] = x)
                 /\   (last (h::t) = last t)`;
(*  If you put the first equation last, the definition does not
work, (Hol bug????) *)
val drop_def = Define `((drop (SUC n) (h::t)) = (drop n t))
                 /\    (drop ZERO xs = xs)
                 /\    (drop n NIL = NIL)
                 /\    (drop 0 NIL = NIL)` handle e => raise e;
```

Note that init and last are partial functions

- Some curious features of the Hol output

```
> val init_def =
    |- (init [x] = []) /\ (init (h::v2::v3) = h::init (v2::v3))
    : Thm.thm
- <<HOL message: inventing new type variable names: 'a.>>
Equations stored under "last_def".
Induction stored under "last_ind".
> val last_def =
    |- (last [x] = x) /\ (last (h::v2::v3) = last (v2::v3))
    : Thm.thm
- <<HOL message: inventing new type variable names: 'a.>>
```

There are overlapping patterns in the Haskell definitions for init and last, and this becomes a concern for theorem proving in Hol

A first proof

An experience with Hol

Haskell version: if b /= [] then last (a ++ b) = last b

```
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !a b. ~(b = []) ==> (last (APPEND a b) = last b)
1 subgoal:
```

REPEAT STRIP_TAC

```
> val it =
    last (APPEND a b) = last b
    ------------------------------------
      ~(b = [])
```

Induct_on 'a'

```
2 subgoals:
> val it =
    !h. last (APPEND (h::a) b) = last b
    ------------------------------------
      0.  ~(b = [])
      1.  last (APPEND a b) = last b

    last (APPEND [] b) = last b
    ------------------------------------
      ~(b = [])
```

PROVE_TAC [APPEND]

```
Goal proved.
 [.] |- last (APPEND [] b) = last b

Remaining subgoals:
> val it =
    !h. last (APPEND (h::a) b) = last b
    ------------------------------------
      0.  ~(b = [])
      1.  last (APPEND a b) = last b

1 subgoal:
```

ASM_REWRITE_TAC [APPEND]

```
> val it =
    !h. last (h::APPEND a b) = last b
    ------------------------------------
      0.  ~(b = [])
      1.  last (APPEND a b) = last b
```

'?x y. APPEND a b = x :: y' by PROVE_TAC[NOT_NULL_STRUCT,APPEND,APPEND_eq_NIL]

```
    1 subgoal:
> val it =
    !h. last (h::APPEND a b) = last b
    ------------------------------------
      0.  ~(b = [])
      1.  last (APPEND a b) = last b
      2.  APPEND a b = x::y

1 subgoal:
```

ASM_REWRITE_TAC [last_DEF]

```
> val it =
    last (x::y) = last b
    ------------------------------------
      0.  ~(b = [])
      1.  last (APPEND a b) = last b
      2.  APPEND a b = x::y
```

PROVE_TAC []

```
Goal proved.
 [...] |- last (x::y) = last b

Goal proved.
 [...] |- !h. last (h::APPEND a b) = last b

Goal proved.
 [..] |- !h. last (h::APPEND a b) = last b

Goal proved.
 [..] |- !h. last (APPEND (h::a) b) = last b

Goal proved.
 [.] |- last (APPEND a b) = last b
> val it =
    Initial goal proved.
    |- !a b. ~(b = []) ==> (last (APPEND a b) = last b)
```

With the Haskell version (last h:t = last t) we could rewrite this easily. With the Hol version (last (h :: v2 :: v3) = last (v2 :: v3)) there is a little more work to do.

An oversight

- Using “Induct_on g” for g of type graph resulting in “ill formed induction theorem”
- I did not know about the following theorem:

```
val graphInduct = TypeBase.induction_of (valOf (TypeBase.read
"RecursiveGraph"));

- > val graphInduct =
    |- !P0 P1 P2 P3 P4.
         (!N a. P0 (SimpleNode N a)) /\
         (!R. P3 R ==> !a N. P0 (RecursiveNode N R a)) /\
         (!N N0 N1 N2 b. P1 (RecursiveEdge N N0 N1 N2 b)) /\
         (!l R l0. P4 l /\ P0 R /\ P4 l0 ==> P2 (RecursiveContext l R l0)) /\
         P3 EmptyRecursiveGraph /\
         (!R R0. P3 R /\ P2 R0 ==> P3 (RecursiveGraph R R0)) /\ P4 [] /\
         (!R l. P1 R /\ P4 l ==> P4 (R::l)) ==>
         (!R. P0 R) /\ (!R. P1 R) /\ (!R. P2 R) /\ (!R. P3 R) /\ !l. P4 l
    : Thm.thm
```

An oversight

- I spent most of my time proving my own well founded induction theorem for graphs
- The "recursive edge problem" came into play
  - Could not easily prove that:
    - deepenEdge (flattenEdge c e) = (deepenEdge o (flattenEdge c)) e
  - One side of the equation got typed as (‘a, ‘b) RecursiveEdge, while the other side got typed as (‘a, ‘c) RecursiveEdge

A better proof

- Fixed the RecursiveEdge problem, by putting RecursiveEdge in a separate Hol_datatype declaration
- Used the induction theorem for graphs provided by Hol, eliminating the need for well founded induction proof
- Resulting .sml file is half as long as the first proof

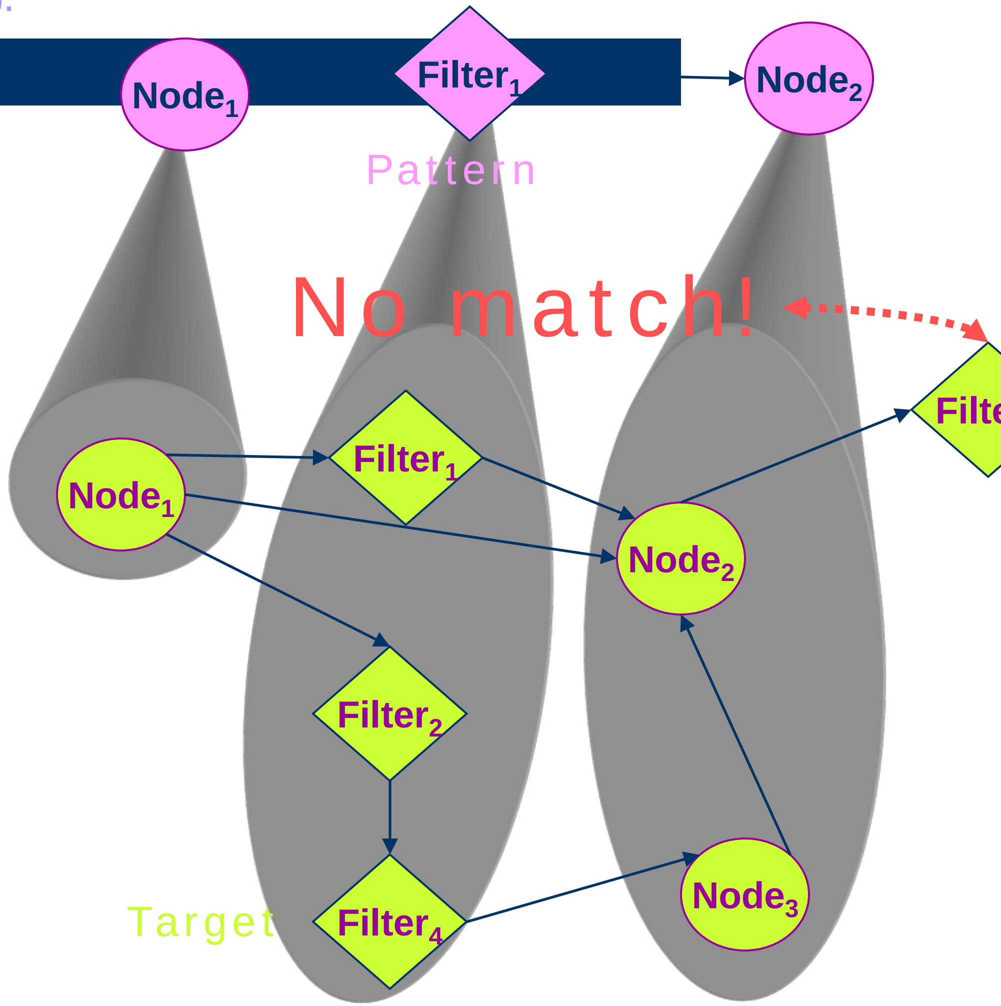
Krenz System Site

Covers and matching rules

Krenz Subgraph / matching

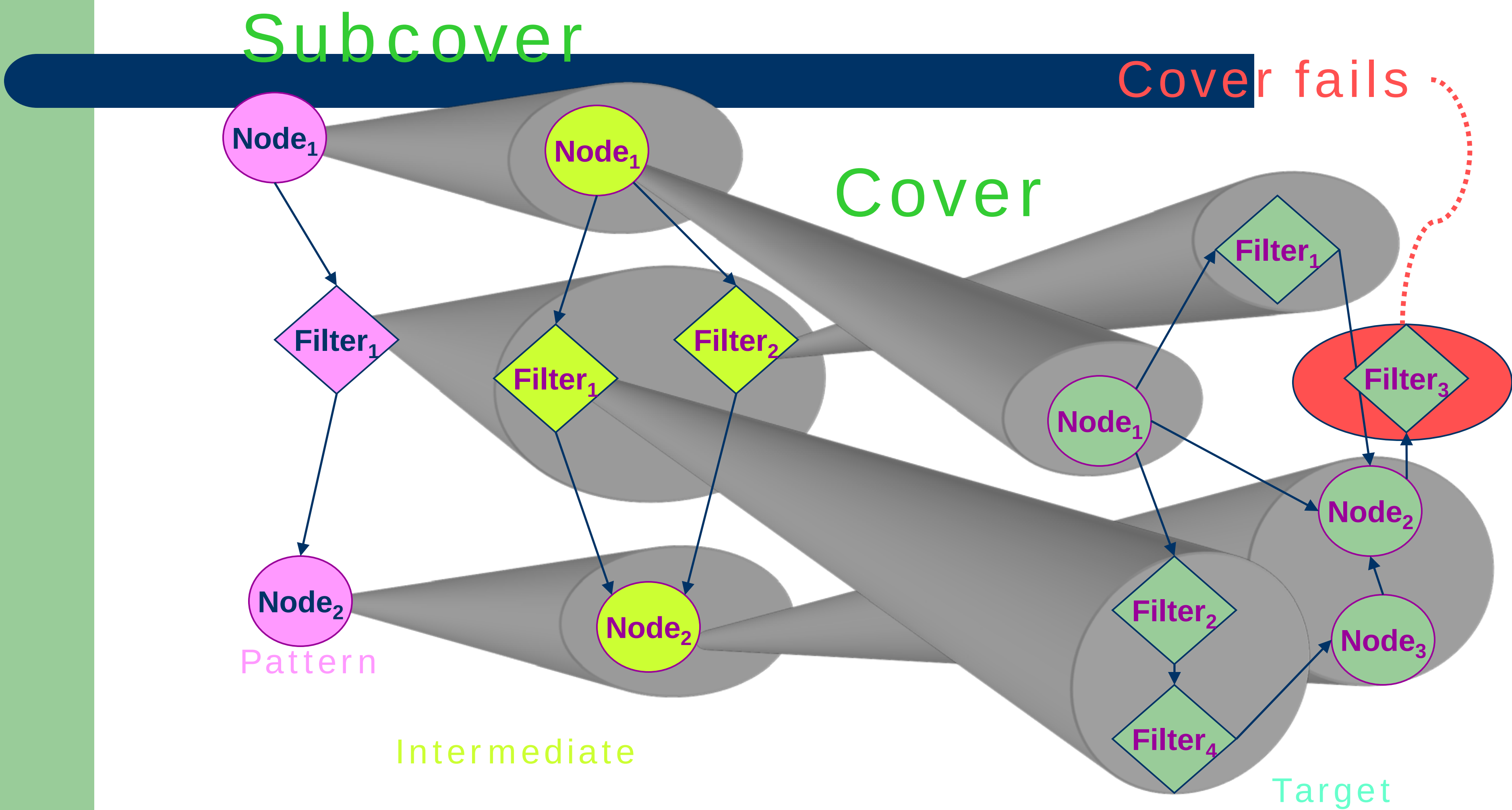
Question: Is the more complicated Krenz system (in green) an instance of the simpler Krenz system (in Lilac).
Alternative formulation: Does the complicated Krenz system (in green) adhere to the Krenz policy (in Lilac).

- Node n can match collection of nodes [m], together with the edges between nodes in [m]
- When n matches N, and m matches M, then the edge (n,m) can match all the edges from N to M
- Node cannot match filter
- Filter f from n to m can match sequence of filters F from N to M if the I/O property of the sequence F is stronger than (implies) the I/O property of the filter f
- Filter f from n to m can match several filters from N to M (same restriction as above)

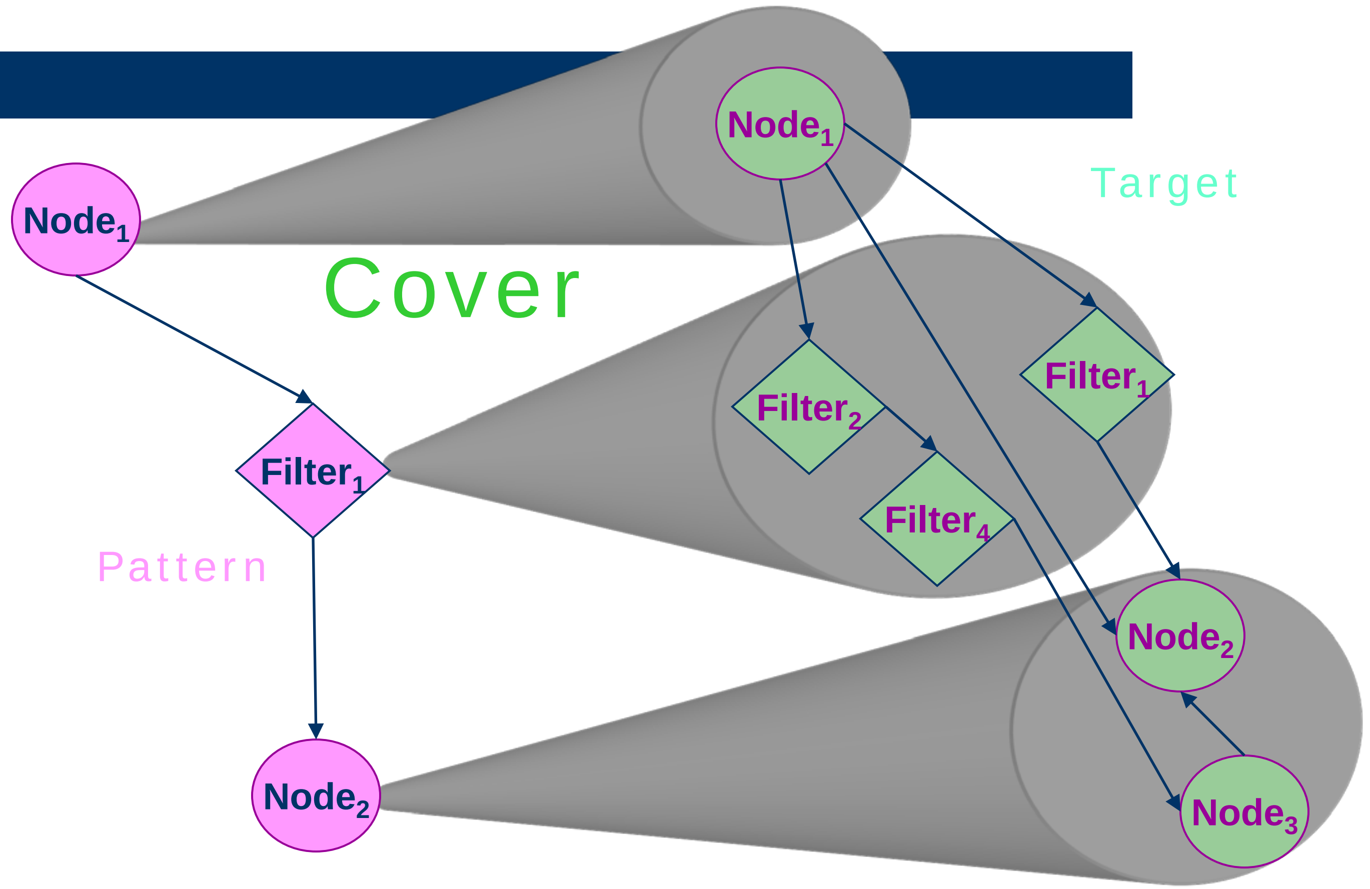


Composition of covers

Notice the concept of a recursive graph arising naturally when the question of adherence to a Krenz policy is raised



Composed cover (Filter_3 deleted)



Grothendieck topology

axioms

Axioms for a Grothendieck topology

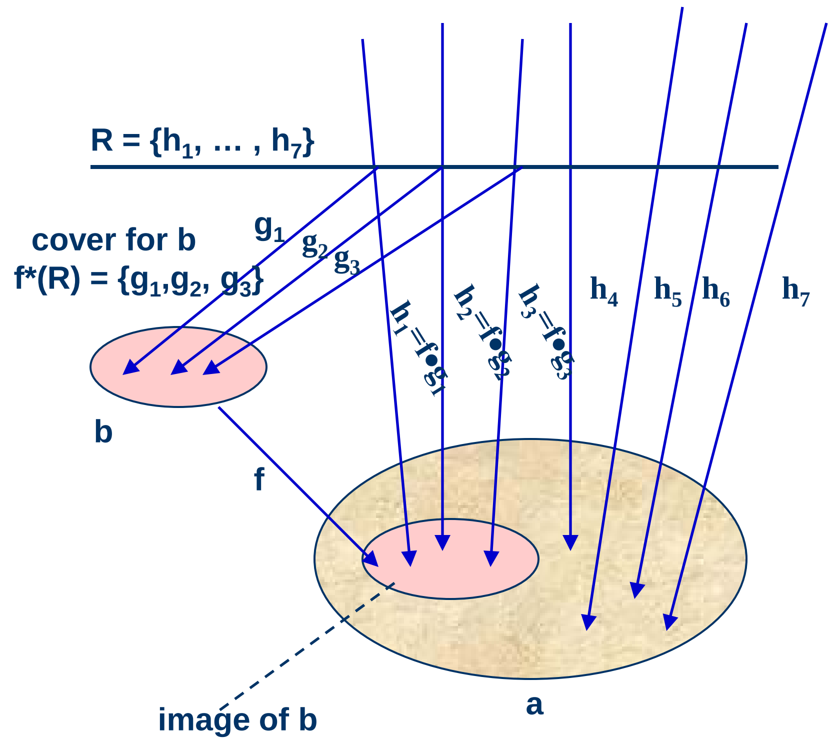
- Stated as properties in tool 0
- A Grothendieck topology *J* on a category *C* is an assignment to each object a of *C*, a set *J(a)* of sieves on a, called covering sieves (or just covers), such that:

Axiom 1: Identity Cover

- Identity Cover:
  - For any object a, the maximal sieve *{f | cod(f) = a}* is in *J(a)*

Axi

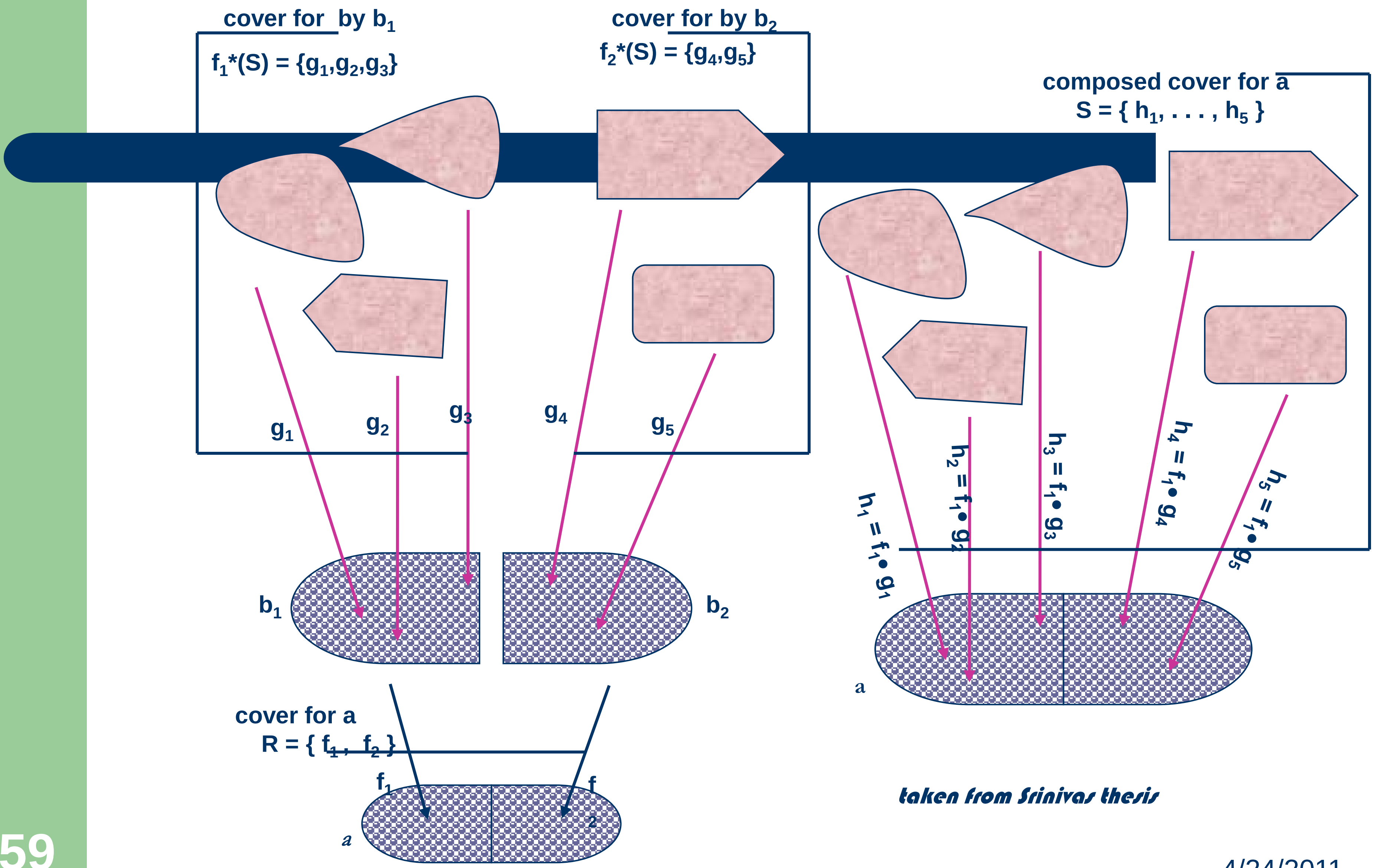
If R is in $J(a)$, and $f::b \rightarrow a$ is an arrow of C, then the sieve $f^*(R) = \{g::c \rightarrow b \mid f \,.\, g \text{ is in } R\}$ is in $J(b)$



taken from Srinivas thesis

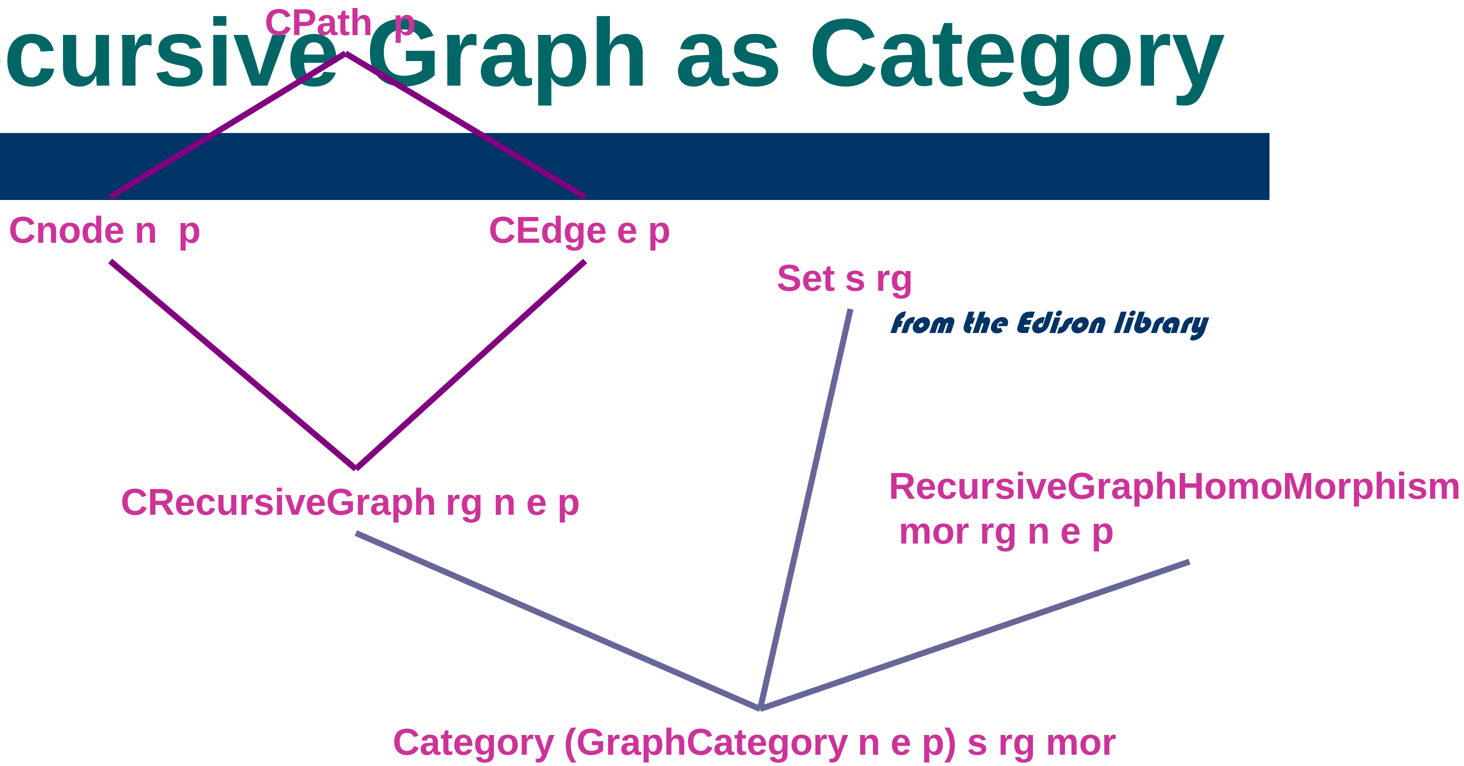
Axiom 3: Stability under refinement

If *R* is in *J(a)* and *S* is a sieve on a such that for each arrow $f::b \rightarrow a$ in *R*, if $f^*(S)$ in $J(b)$, then *S* is in $J(a)$



taken from Srinivas thesis

Recursive Graph as Category



To use all of this machinery requires placing the recursive graph in the setting of category theory