EJCP 2016

Model Checking Modulo Theories with Cubicle

Sylvain Conchon

LRI (UMR 8623), Université Paris-Sud
Équipe Toccata, INRIA Saclay – Île-de-France

Cubicle

An SMT based model checker for parameterized systems

Contents

- A short tutorial on Cubicle
- Theoretical foundations
- Implementation details

Cubicle : an open source model checker

Cubicle is an open source software written in OCaml.

- It is based on the theoretical framework of Model Checking Modulo Theories [Ghilardi, Ranise]
- Its implementation relies on a lightweight and enhanced version of the SMT solver Alt-Ergo

Cubicle is the result of a fruitful collaboration between University of Paris-Sud and Intel corporation

Cubicle's input language

Inspired by the description language of the Murφ model checker (Dill's group at Stanford)

A program is only represented by a set of global variables and guarded commands.

Guarded commands

A guarded command consists of a predicate on the state variables, called a guard, and a set of assignments that update those variables to change the state.

The control structure is only a single infinite loop which repeatedly execute two steps:

1. evaluate all the guards, given the current values of global variables
2. arbitrarily choose one of the commands whose guard is true and execute it, updating the variables

Example 1 : sequential programs

```
X,Y = 0

L0:
    X := 1;
L1:
    Y := X + 1;
L2:
```

```
type loc = L0 | L1 | L2
var X : int
var Y : int
var PC : loc

init () { X = 0 && Y = 0 && PC = L0 }

transition t1 ()
requires { PC = L0 }
{ X := 1; PC := L1  }

transition t2 ()
requires { PC = L1 }
{ Y := X + 1; PC := L2  }
```

Example 2 : granularity

```
L0:
  X := 1;
L1:
  Y := X + 1;
L2:
```

Example 2 : granularity

```
L0:
  X := 1;
L1:
  EAX := X;
L2:
  EAX := EAX + 1;
L3:
  Y := EAX;
L4:
```

Example 2 : granularity

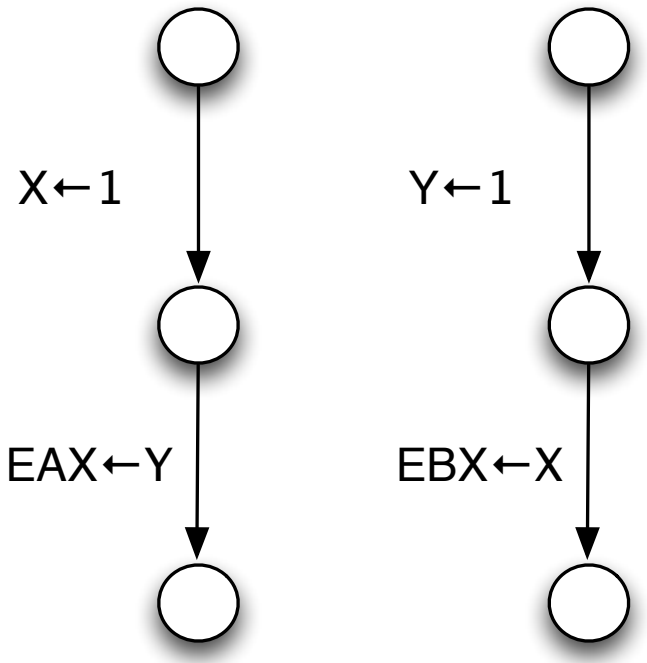
```
L0:
  X := 1;
L1:
  EAX := X;
L2:
  EAX := EAX + 1;
L3:
  Y := EAX;
L4:
```

```
type loc = L0 | L1 | L2 | L3 | L4
var X : int
var Y : int
var EAX : int
var PC : loc

init () { X = 0 && Y = 0 && PC = L0 }

transition t1 ()
requires { PC = L0 }{ X := 1; PC := L1  }

transition t2 ()
requires { PC = L1 }{ EAX := X; PC := L2  }

transition t3 ()
requires { PC = L2 }
{ EAX := EAX + 1; PC := L3  }

transition t4 ()
requires { PC = L3 }{ Y := EAX; PC := L4  }
```

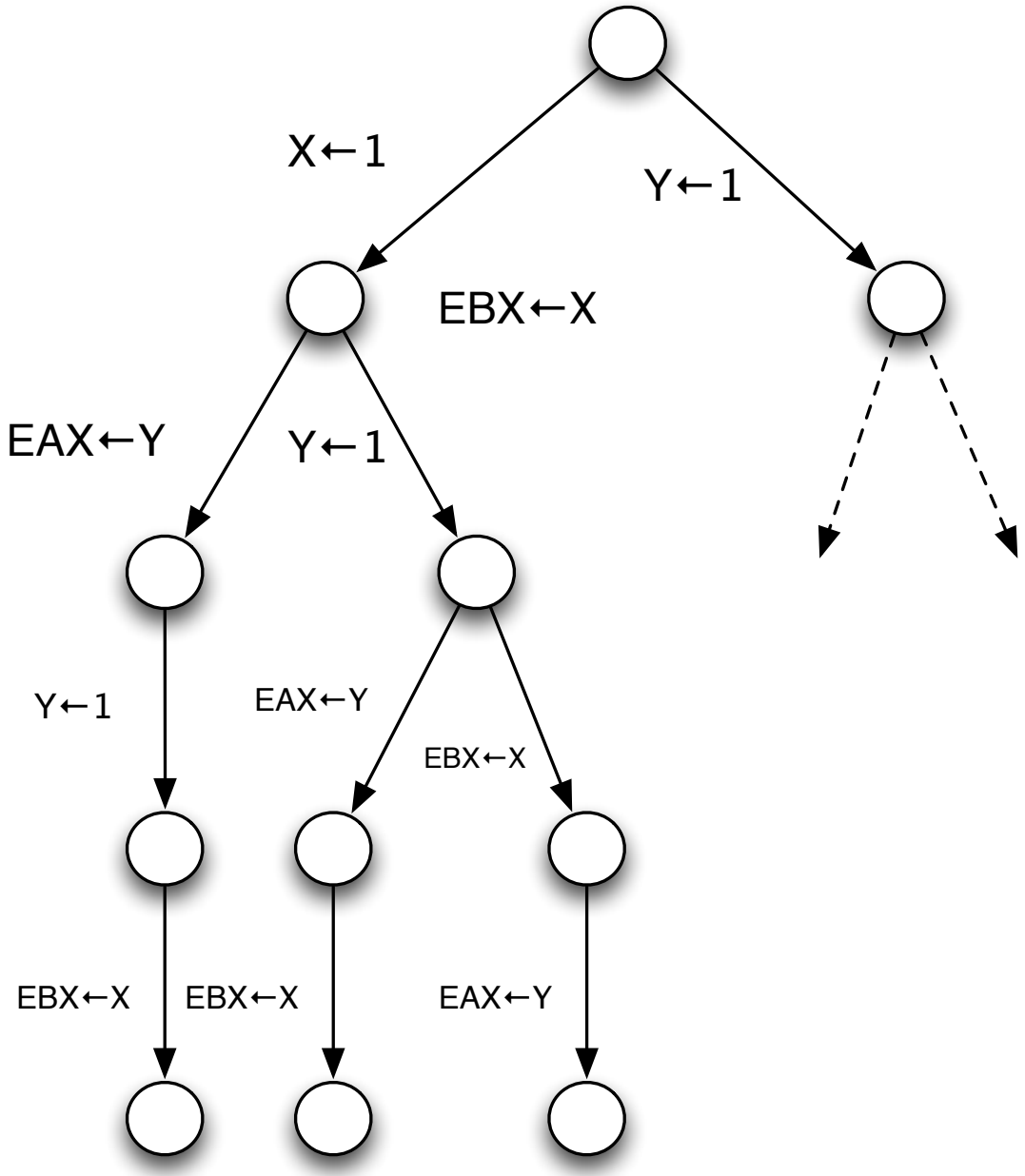
Example 3 : threads

X,Y = 0

Thread 1	Thread 2
L0:	L0:
X := 1;	Y := 1;
L1:	L1:
EAX := Y;	EBX := X;
L2:	L2:



Example 3 : interleaving semantics



Example 3

```
type location = L0 | L1 | L2
var X : int
var Y : int
var EAX : int
var EBX : int
var PC1 : location
var PC2 : location

init () { PC1 = L0 && PC2 = L0 && X = 0 && Y = 0 }

transition writex_1 ()
requires { PC1 = L0 } { X := 1; PC1 := L1 }

transition ready_1 ()
requires { PC1 = L1  } { EAX := Y; PC1 := L2 }

transition writey_2 ()
requires { PC2 = L0 } { Y := 1; PC2 := L1 }

transition readx_2 ()
requires { PC2 = L1 } { EBX := X; PC2 := L2 }
```

Example 3 : safety property

- Characterize bad states
- Use Cubicle to show that bad states cannot be reach from initial states

```
type loc = L0 | L1 | L2
var X : int
var Y : int
var EAX : int
var EBX : int
var PC1 : loc
var PC2 : loc
init () { PC1 = L0 && PC2 = L0 && X = 0 && Y = 0 }
unsafe () { PC1 = L2 && PC2 = L2 && EAX = 0 && EBX = 0 }
...
```

Parameterized systems

Modeling and verifying programs involving an arbitrary number of processes

- Replication of components
- Unknown or very large number of components

Typical examples : cache coherence protocols, mutual exclusion algorithms, fault-tolerant protocols

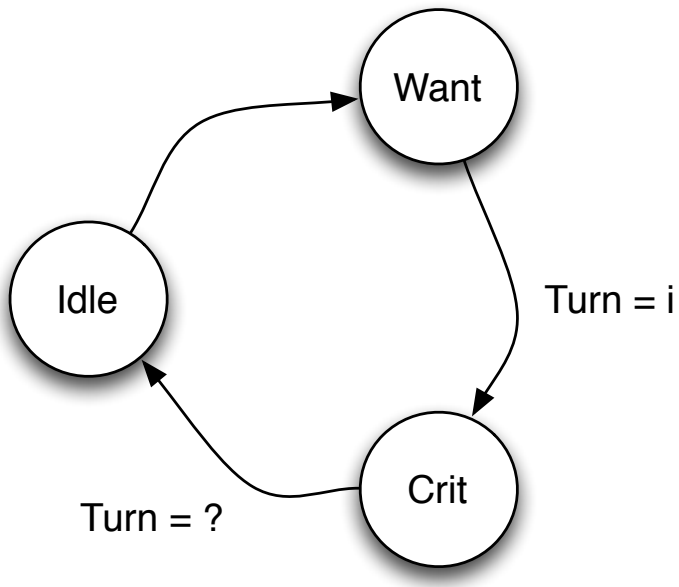
Array-based transition systems

Cubicle handles parameterized systems through :

- a new built-in data type proc (with unspecified cardinality)
- state variables defined as arrays indexed by process identifiers
- initial states described with a universally-quantified formula over processes
- bad states described with existentially-quantified formulas over processes
- transitions parameterized by process identifiers

Example 4 : array-based transition systems

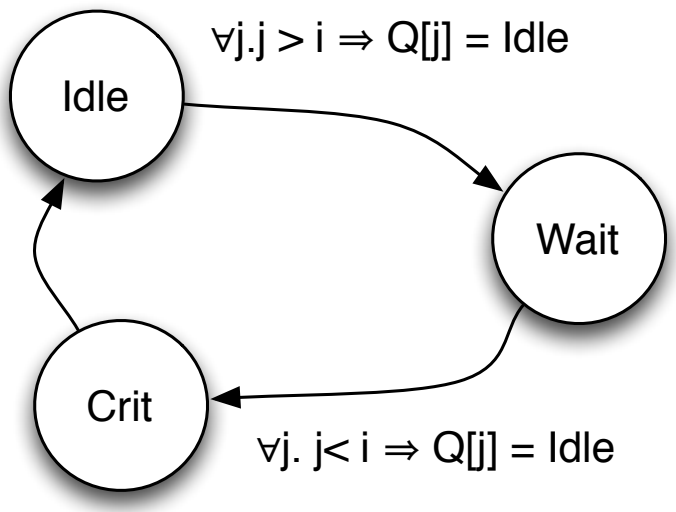
A Dekker-like mutual exclusion algorithm




```
type st = Idle | Want | Crit
var Turn : proc
array S[proc] : st

init (z) { S[z] = Idle }

unsafe (x y)
  { S[x] = Crit && S[y] = Crit }

transition req (i)
requires { S[i] = Idle }
{ S[i] := Want  }

transition enter (i)
requires { S[i] = Want && Turn = i }
{ S[i] := Crit }

transition exit (i)
requires { S[i] = Crit }
{ Turn := ? ; S[i] := Idle }
```

Example 5 : global conditions

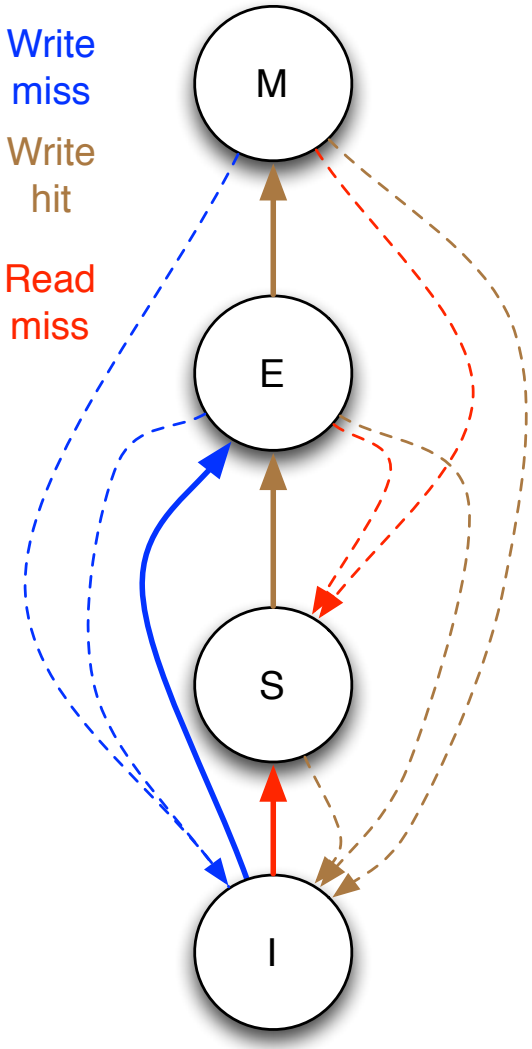
A Bakery-like mutual exclusion algorithm




```
type t = Idle | Wait | Crit
array Q[proc] : t

init (i) { Q[i]=Idle }
unsafe (i j) { Q[i]=Crit && Q[j]=Crit }

transition wait (i)
requires { Q[i]=Idle &&
           forall_other j. (j>i => Q[j]=Idle) }
{ Q[i] := Wait }

transition enter (i)
requires { Q[i]=Wait &&
           forall_other j. (j<i => Q[j]=Idle) }
{ Q[i] := Crit }

transition exit (i)
requires { Q[i] = Crit }
{ Q[i] := Idle }
```

Example 6 : case analysis

MESI protocol




```
type state = M | E | S | I
array St[proc]: state
init (i) { St[i] = I }

transition read_miss ( i )
requires { St[i] = I }
{ St[j] := case  | j = i : S
                 | St[j] = E : S
                 | St[j] = M : S
                 | _ : St[j]      }

transition write_miss ( i )
requires { St[i] = I }
{ St[j] := case | j = i : E | _ : I }

transition write_hit_1 ( i )
requires { St[i] = E }{ St[i] := M }

transition write_hit_2 ( i )
requires { St[i] = S }
{ St[j] := case | j = i : E | _ : I }
```

Language limitations

Cubicle's input language is still limited; it could be further improved.

- data types (records, several parameterized data types)
- programming constructs (sequences, loops)
- arithmetic expressions

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i)` { `A[i] = True && PC = L1` }

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i) { A[i] = True && PC = L1 }`
  $\forall i : \texttt{proc.A}[i] \wedge \texttt{PC} = \texttt{L1}$

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i) { A[i] = True && PC = L1 }`
  $\forall i : \texttt{proc.A}[i] \wedge \texttt{PC} = \texttt{L1}$
- Bad states are defined by special existentially quantified formulas, called cubes

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i) { A[i] = True && PC = L1 }`
  $\forall i : \texttt{proc.A}[i] \wedge \texttt{PC} = \texttt{L1}$
- Bad states are defined by special existentially quantified formulas, called cubes
  unsafe `(i j) { S[i] = Crit && S[j] = Crit }`

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i) { A[i] = True && PC = L1 }`
  $\forall i : \texttt{proc}.\texttt{A}[i] \wedge \texttt{PC} = \texttt{L1}$
- Bad states are defined by special existentially quantified formulas, called cubes
  unsafe `(i j) { S[i] = Crit && S[j] = Crit }`
  $\exists i, j : \texttt{proc}.\, i \neq j \wedge \texttt{S[i]} = \texttt{Crit} \wedge \texttt{S[j]} = \texttt{Crit}$

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula
  init `(i) { A[i] = True && PC = L1 }`
  $\forall i : \texttt{proc}.\texttt{A}[i] \wedge \texttt{PC} = \texttt{L1}$
- Bad states are defined by special existentially quantified formulas, called cubes
  unsafe `(i j) { S[i] = Crit && S[j] = Crit }`
  $\exists i, j : \texttt{proc}.\, i \neq j \wedge \texttt{S[i]} = \texttt{Crit} \wedge \texttt{S[j]} = \texttt{Crit}$
- Transitions correspond to existentially quantified formulas

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula

  init `(i) { A[i] = True && PC = L1 }`

  $\forall i : \texttt{proc.A}[i] \wedge \texttt{PC} = \texttt{L1}$

- Bad states are defined by special existentially quantified formulas, called cubes

  unsafe `(i j) { S[i] = Crit && S[j] = Crit }`

  $\exists i, j : \texttt{proc.}\, i \neq j \wedge \texttt{S[i]} = \texttt{Crit} \wedge \texttt{S[j]} = \texttt{Crit}$

- Transitions correspond to existentially quantified formulas

  transition `t(i)`
  requires `{ S[i] = A && PC = L1 }`
  `{ S[i] = B; X = X+1 }`

Theoretical foundation : MCMT

Cubicle implements the Model Checking Modulo Theories (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- Initial states are defined by a universally quantified formula

  init (i) { A[i] = True && PC = L1 }

  $\forall i : \texttt{proc}.\texttt{A}[i] \wedge \texttt{PC} = \texttt{L1}$

- Bad states are defined by special existentially quantified formulas, called cubes

  unsafe (i j) { S[i] = Crit && S[j] = Crit }

  $\exists i, j : \texttt{proc}.\, i \neq j \wedge \texttt{S}[\texttt{i}] = \texttt{Crit} \wedge \texttt{S}[\texttt{j}] = \texttt{Crit}$

- Transitions correspond to existentially quantified formulas

  transition t(i)
  requires { S[i] = A && PC = L1 }
  { S[i] = B; X = X+1 }

  $\exists i : \texttt{proc}.\, \texttt{S}[\texttt{i}] = \texttt{A} \wedge \texttt{PC} = \texttt{L1} \wedge \texttt{S'} = \texttt{S}[\texttt{i} \texttt{<-} \texttt{B}] \wedge \texttt{X'} = \texttt{X} + 1$

Inductive invariants

We are looking for a predicate Reach such that :

Reach is an inductive invariant :

$$\forall \vec{x}.\mathsf{Init}(\vec{x}) \Rightarrow \mathsf{Reach}(\vec{x})$$

$$\forall \vec{x}, \vec{x}'.\mathsf{Reach}(\vec{x}) \wedge \tau(\vec{x}, \vec{x}') \Rightarrow \mathsf{Reach}(\vec{x}')$$

The system is safe if there exists an interpretation of Reach such that :

$$\forall \vec{x}.\mathsf{Reach}(\vec{x}) \models \neg\mathsf{unsafe}(\vec{x})$$

Example 7 : Back to Dekker-like algorithm

```
type st = Idle | Want | Crit
var T : proc
array S[proc] : st

init (z) { S[z] = Idle }

unsafe (x y) {
  S[x] = Crit && S[y] = Crit
}

transition req (i)
requires { S[i] = Idle }
{ S[i] := Want  }
```

```
transition enter (i)
requires { S[i] = Want &&
           T = i }
{ S[i] := Crit }

transition exit (i)
requires { S[i] = Crit }
{ T := ? ;
  S[i] := Idle }
```

A pure SMT problem

```
type st = Idle | Want | Crit
type proc
logic Reach : proc, (proc,st) farray -> prop

axiom init :
 forall t:proc. forall s:(proc,st) farray. (forall z:proc. s[z]=Idle) -> Reach(t,s)

axiom req :
 forall t,t':proc. forall s,s':(proc,st) farray.
 Reach(t,s) and ( exists i:proc. s[i]=Idle and s'=s[i<-Want] and t'=t)
 -> Reach(t',s')

axiom enter :
 forall t,t':proc. forall s,s':(proc,st) farray.
 Reach(t,s) and (exists i:proc. s[i]=Want and t=i and s'= s[i<-Crit] and t'=t)
 -> Reach(t',s')

axiom exit :
 forall t,t':proc. forall s,s':(proc,st) farray.
 Reach(t,s) and ( exists i:proc. s[i]=Crit and s'=s[i<-Idle]) -> Reach(t',s')

goal unsafe :
 exists t:proc. exists s:(proc,st) farray. exists i,j:proc.
 i<>j and Reach(t,s) and s[i]=Crit and s[j]=Crit
```

Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
 forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
exists x. Reach(x) and x = 9
```

Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
 forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be $\{0, 2, 4, 6, \ldots\}$

Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
 forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be $\{0, 2, 4, 6, \ldots\}$

or $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \ldots\}$

Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
 forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be $\{0, 2, 4, 6, \ldots\}$

or $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \ldots\}$ but not $\mathbb{N}$ nor $\{0, 2, 4, 5, 6, 7, \ldots\}$

Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
 forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
exists x. Reach(x) and x = 9
```

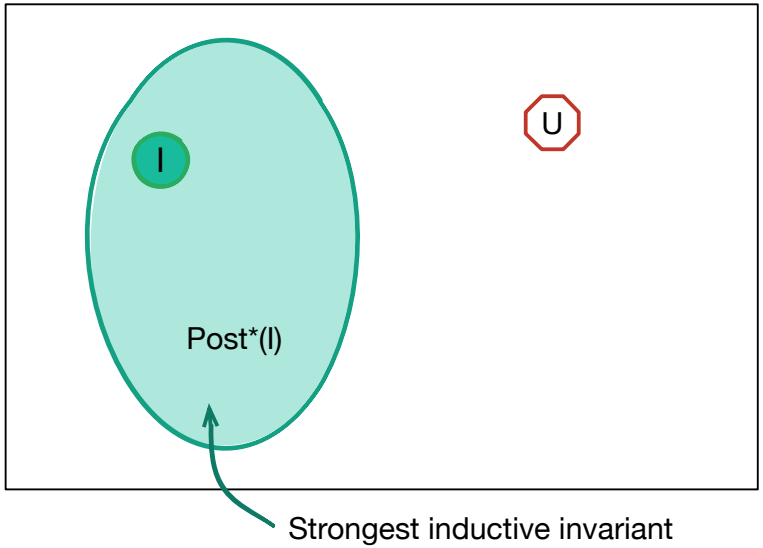
A possible interpretation for Reach could be $\{0, 2, 4, 6, \ldots\}$

or $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \ldots\}$ but not $\mathbb{N}$ nor $\{0, 2, 4, 5, 6, 7, \ldots\}$

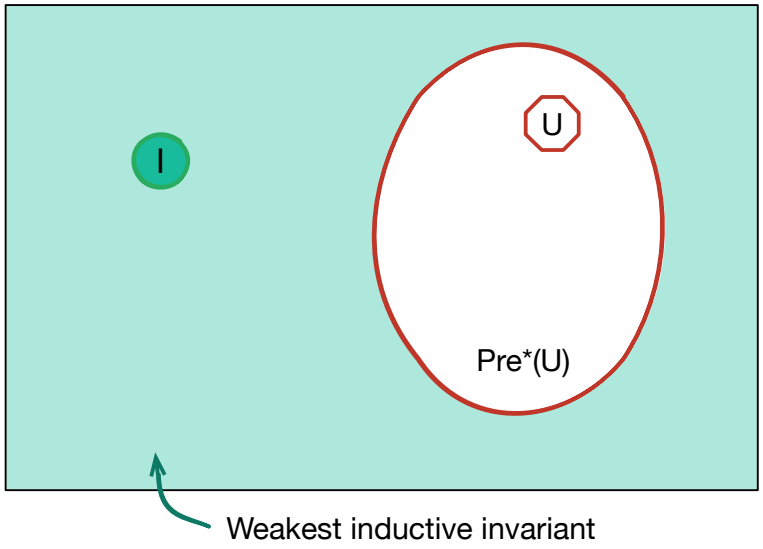
How to find interpretations for inductive invariants ?

Inductive invariants (2)

Set of **reachable** states from Init
= Strongest inductive invariant



Set of states that **cannot reach** Unsafe
= Weakest inductive invariant



Backward Reachability (BR) : How to compute Pre*(U)

I : inital states U : unsafe states (cube) $\mathcal{T}$: transitions

---

BR ():

V := $\emptyset$;

push(Q, U) ;

while not_empty(Q) **do**

φ := pop(Q)

if $\varphi \wedge$ I sat **then return** unsafe

if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ **then**

V := V $\cup \{\varphi\}$

push(Q, $\mathtt{pre}_{\mathcal{T}}(\varphi)$)

return safe

Backward Reachability (BR) : How to compute Pre*(U)

I : inital states U : unsafe states (cube) $\mathcal{T}$: transitions

---

BR ():

V := $\emptyset$;

push(Q, U) ;

while not_empty(Q) **do**

φ := pop(Q)

if $\varphi \wedge$ I sat **then return** unsafe (* SMT check *)

if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ **then** (* SMT check *)

V := V $\cup \{\varphi\}$

push(Q, $\mathtt{pre}_{\mathcal{T}}(\varphi)$)

return safe

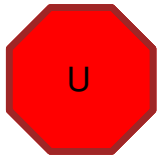
Pre-images

Given a transition $t \in \mathcal{T}$, $\texttt{pre}_t(\varphi)$ is a formula that describes the set of states from which a φ-state is reachable in one t-step

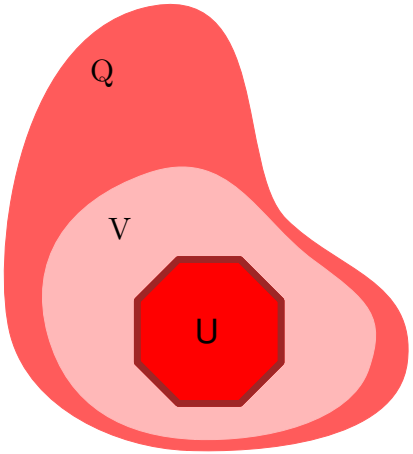
$$\texttt{pre}_{\mathcal{T}}(\varphi) = \bigvee_{t \in \mathcal{T}} \texttt{pre}_t(\varphi)$$

If φ is a cube, then $\texttt{pre}_{\mathcal{T}}(\varphi)$ can also be represented as a union of cubes

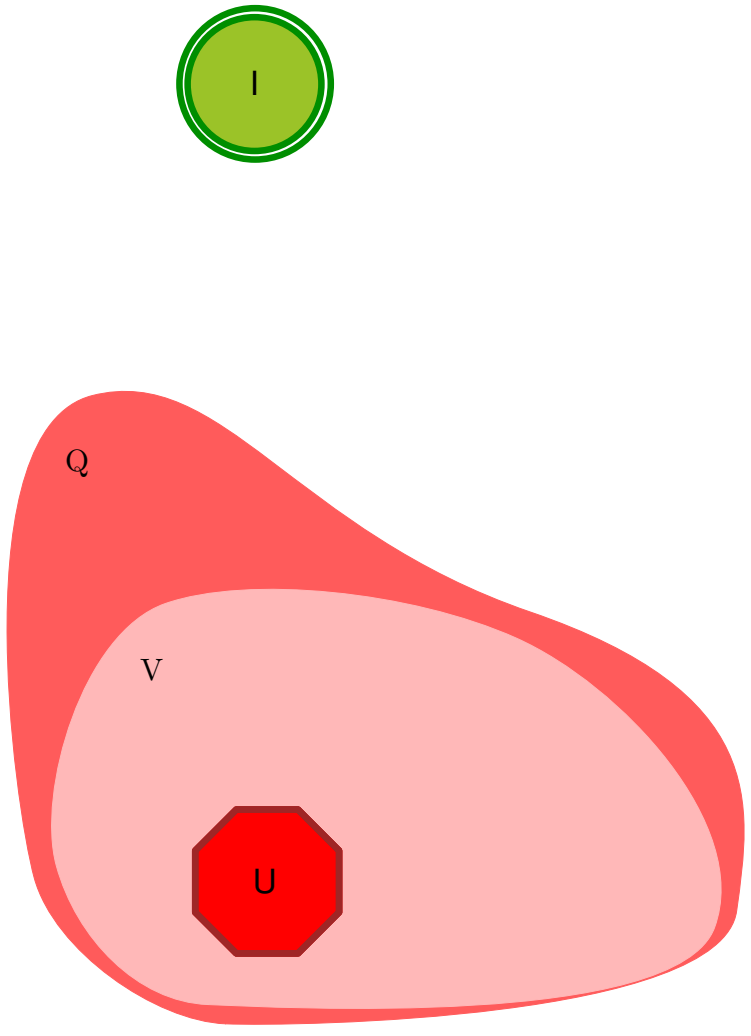
Running BR



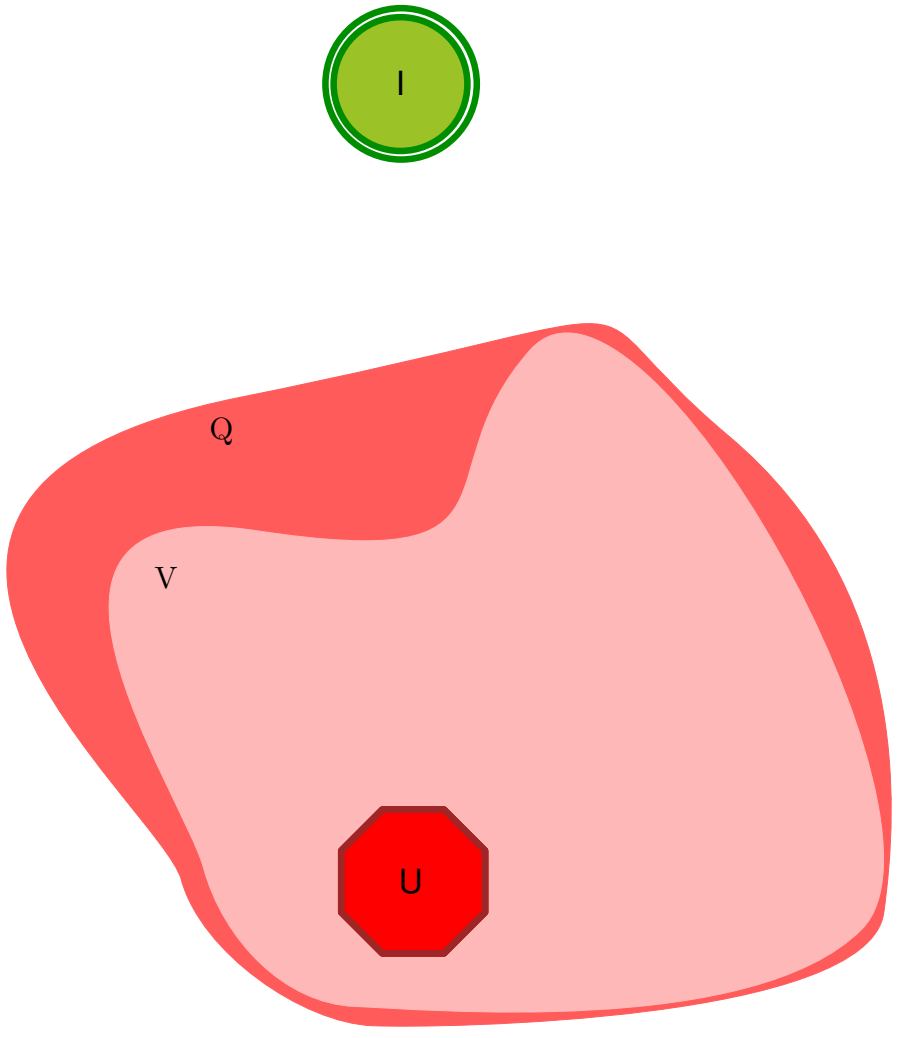
Running BR



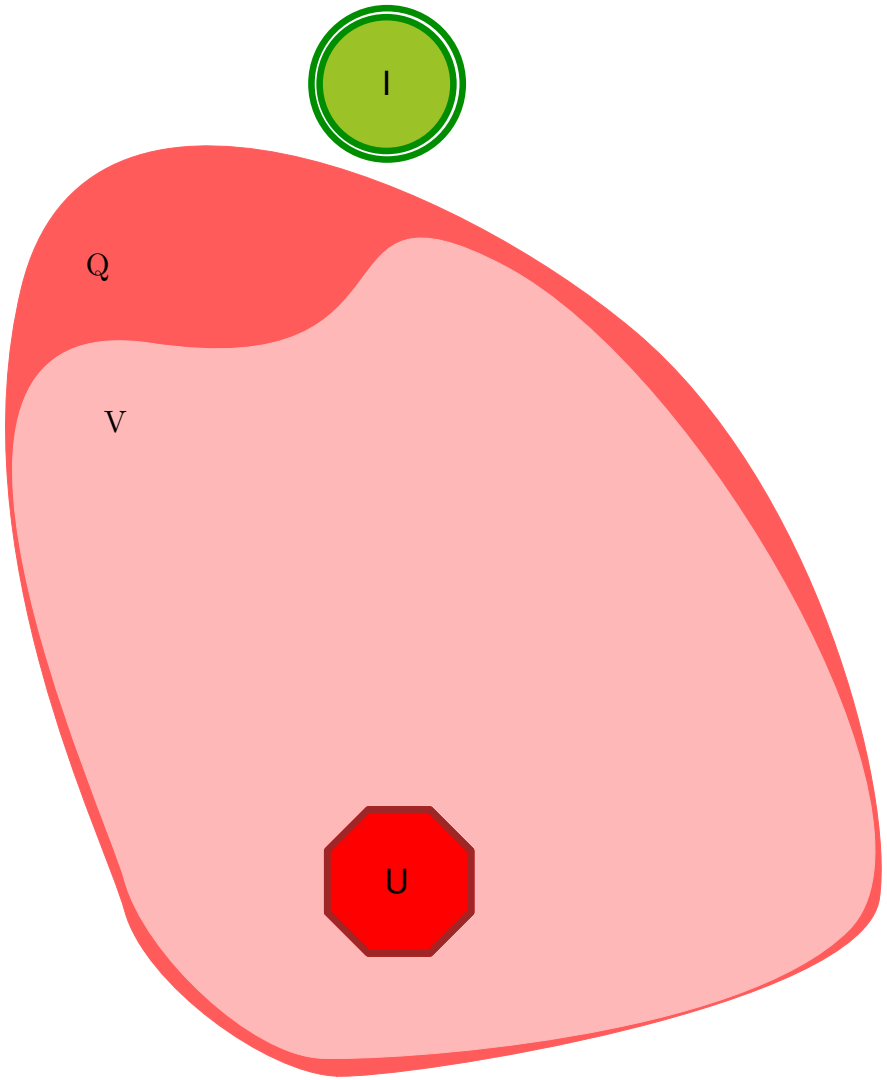
Running BR



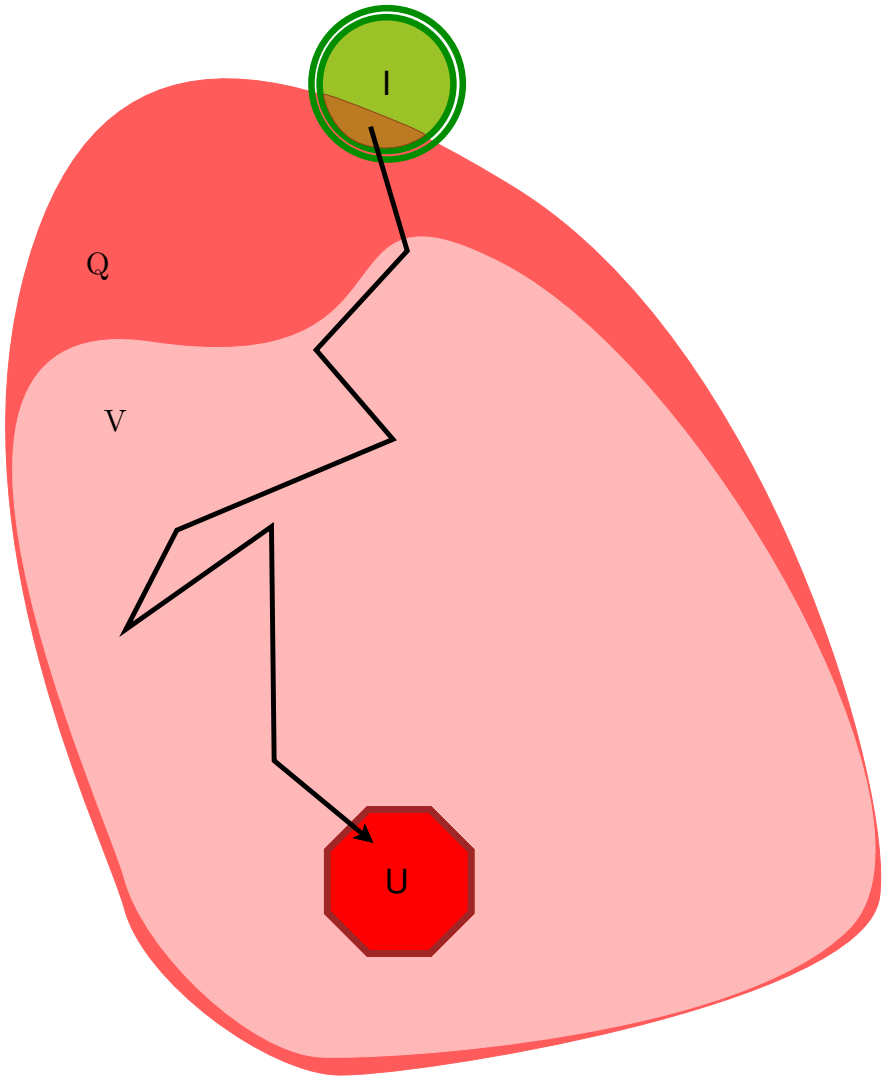
Running BR



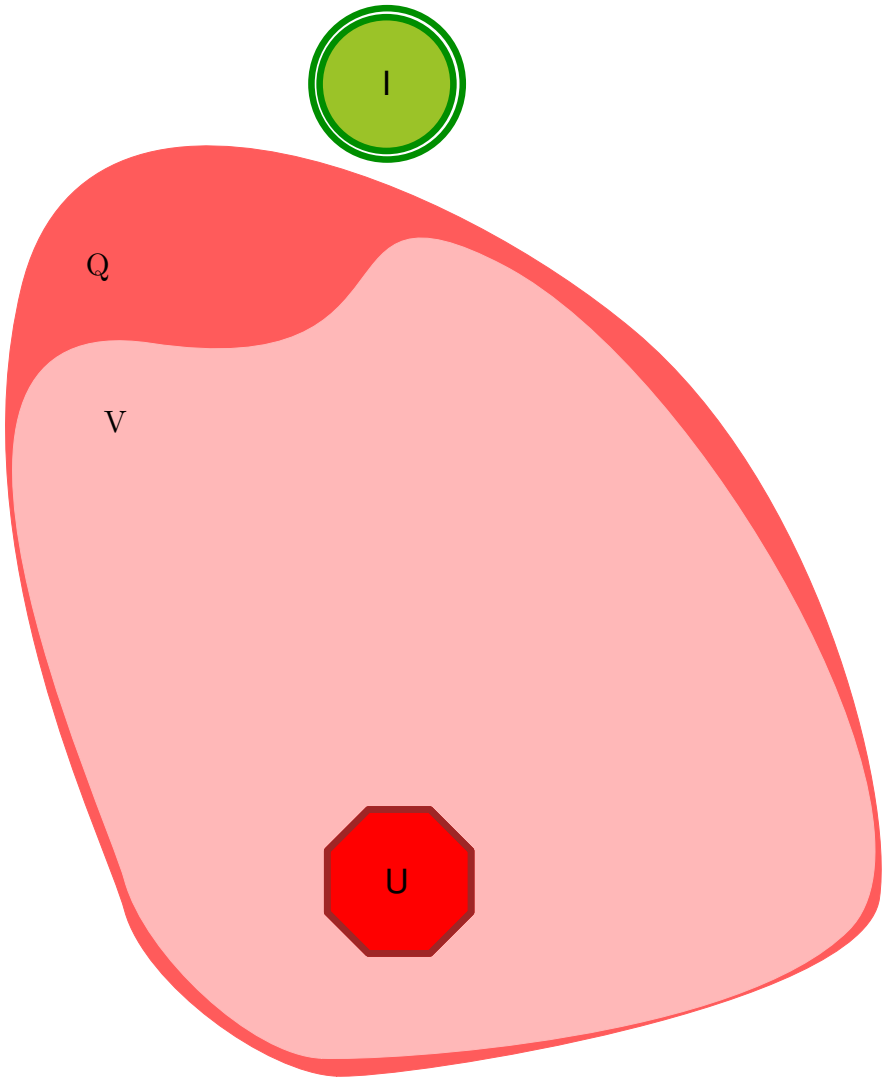
Running BR



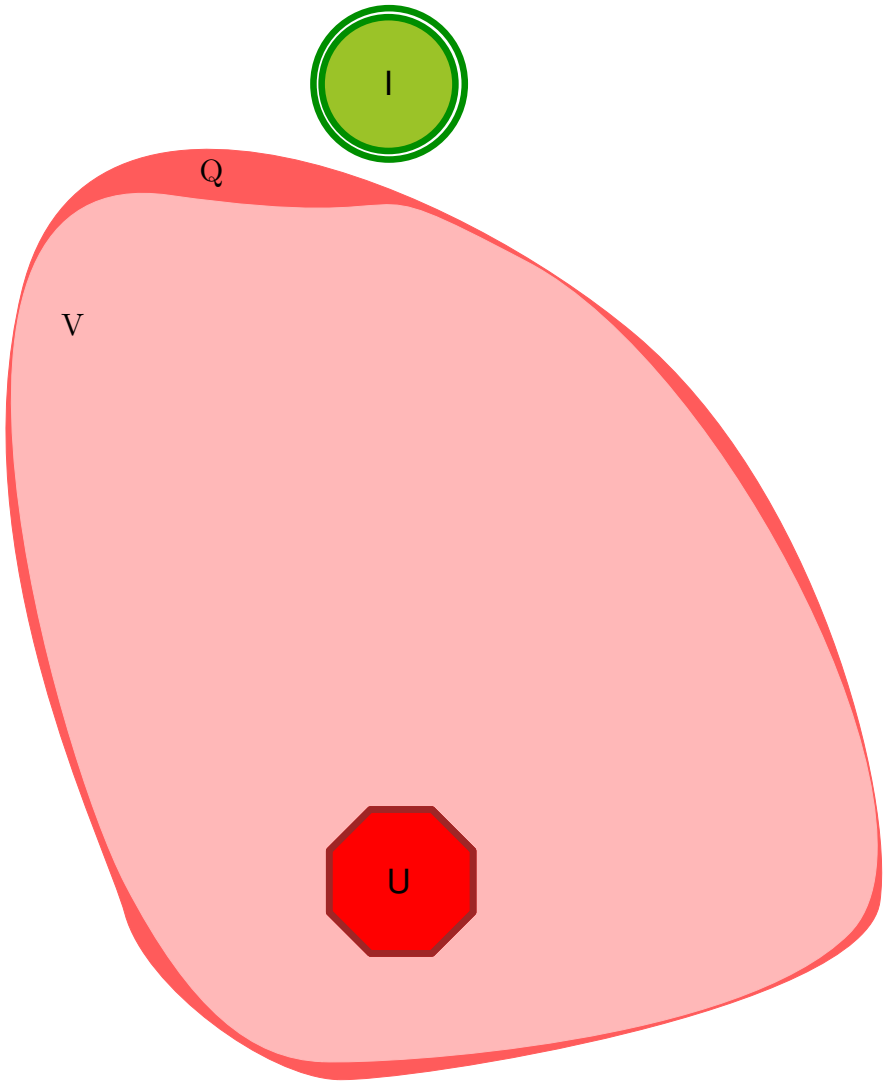
Running BR



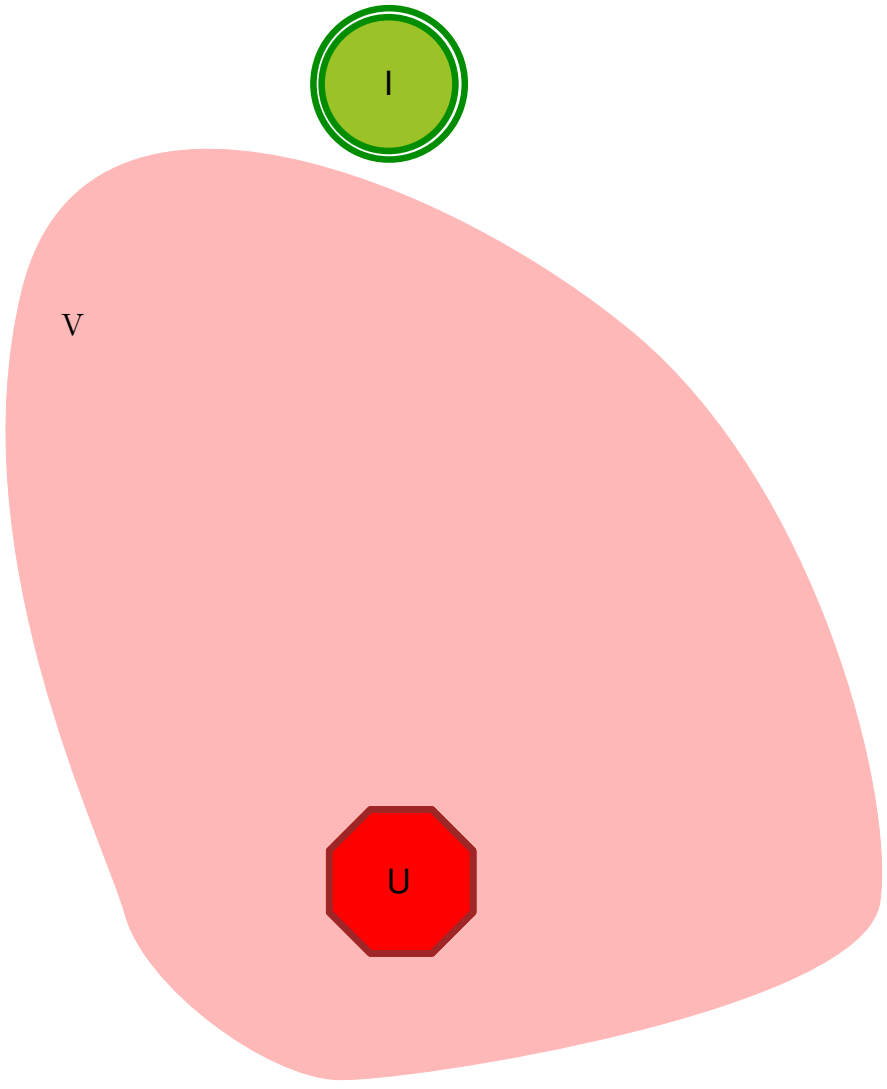
Running BR



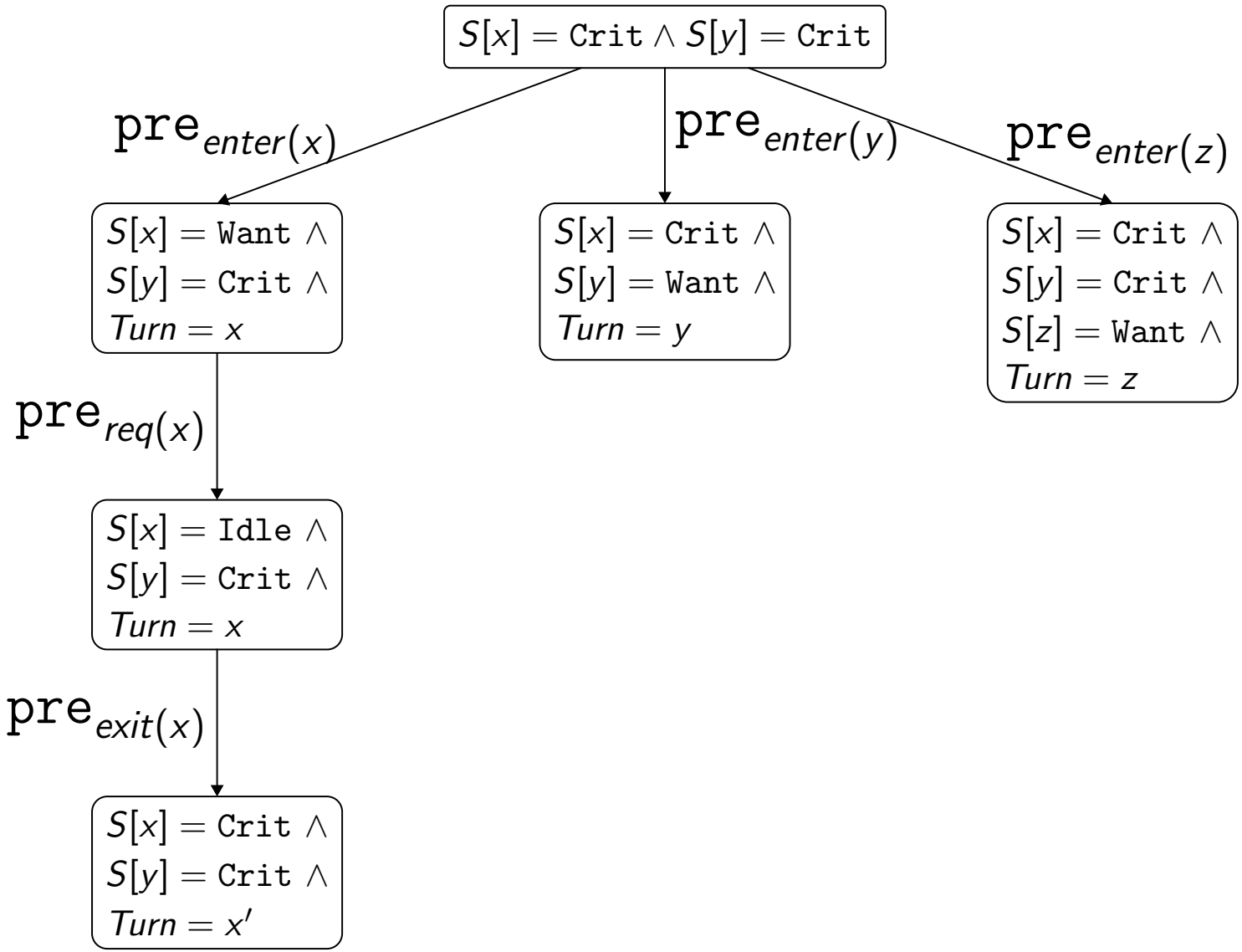
Running BR



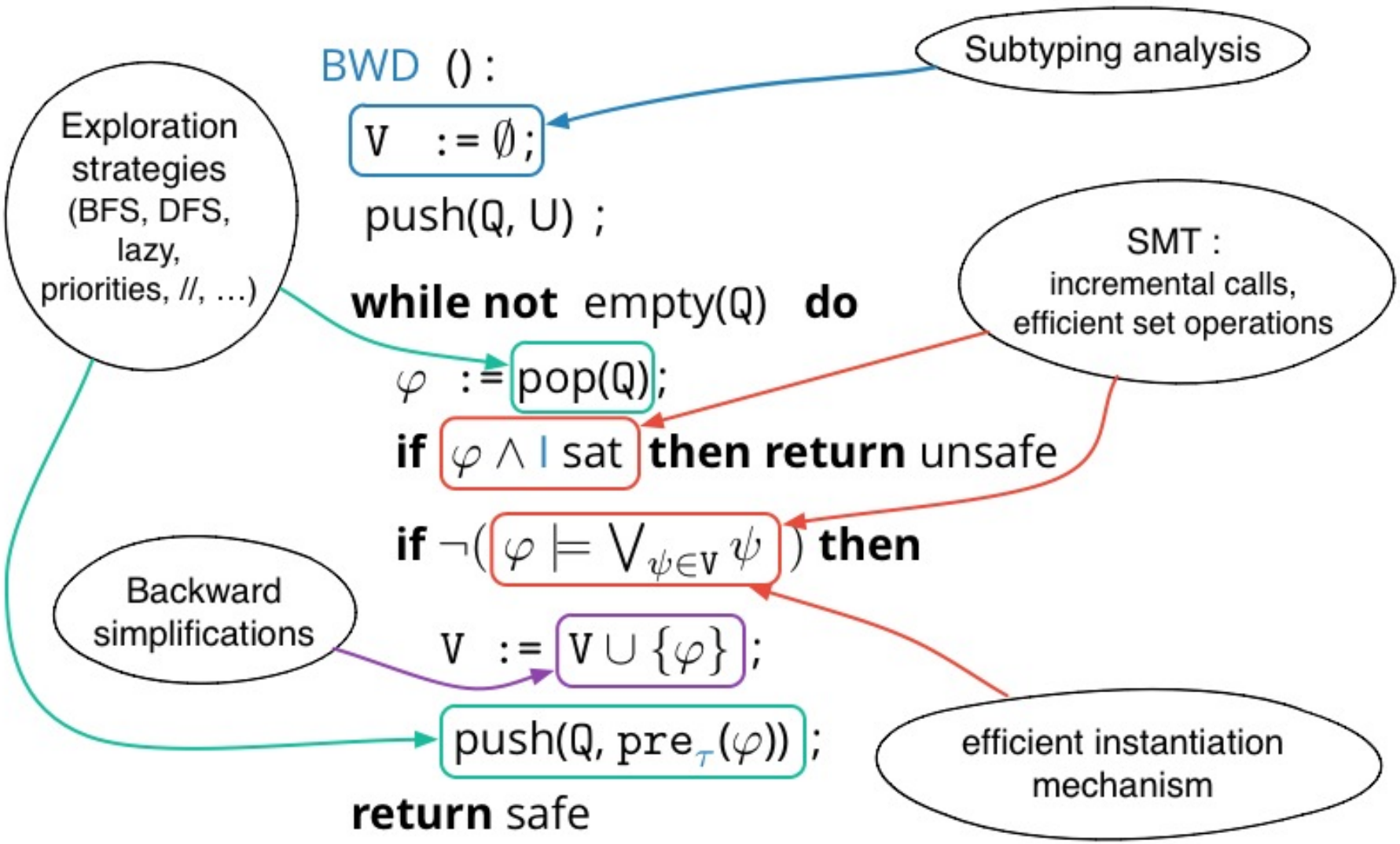
Running BR



BR : example


$S[x] = \texttt{Crit} \wedge S[y] = \texttt{Crit}$
$\texttt{pre}_{enter(x)}$
$\texttt{pre}_{enter(y)}$
$\texttt{pre}_{enter(z)}$
$S[x] = \texttt{Want} \wedge$
$S[y] = \texttt{Crit} \wedge$
$Turn = x$
$S[x] = \texttt{Crit} \wedge$
$S[y] = \texttt{Want} \wedge$
$Turn = y$
$S[x] = \texttt{Crit} \wedge$
$S[y] = \texttt{Crit} \wedge$
$S[z] = \texttt{Want} \wedge$
$Turn = z$
$\texttt{pre}_{req(x)}$
$S[x] = \texttt{Idle} \wedge$
$S[y] = \texttt{Crit} \wedge$
$Turn = x$
$\texttt{pre}_{exit(x)}$
$S[x] = \texttt{Crit} \wedge$
$S[y] = \texttt{Crit} \wedge$
$Turn = x'$

Implementation issues and optimizations



Fixpoint computation

```
V := ∅
push(Q, U)
while not_empty(Q) do
  φ := pop(Q)
  if φ ∧ I sat then return(unsafe)
  if ¬(φ ⊨ ⋁_{ψ∈V} ψ) then
    V := V ∪ {φ}
    push(Q, pre_T(φ))
return(safe)
```

Fixpoint computation: a challenge

$$\varphi \models \bigvee_{\psi \in \mathbf{V}} \psi$$

Fixpoint computation: a challenge

$$\exists \bar{x}.F \models \bigvee_{\psi \in \mathtt{V}} \exists \bar{y}.G_{\psi}$$

Fixpoint computation: a challenge

$$\exists \bar{x}.F \wedge \bigwedge_{\psi \in \mathtt{V}} \forall \bar{y}.\neg G_\psi \qquad \text{satisfiable ?}$$

Fixpoint computation: a challenge

$$\exists \bar{x}.F \wedge \bigwedge_{\psi \in \mathtt{V}} \forall \bar{y}.\neg G_\psi \qquad \text{satisfiable ?}$$

F and G_ψ are conjunctions of literals involving several theories
(uninterpreted function symbols, linear arithmetic, enumerations)

Fixpoint computation: a challenge

$$\exists \bar{x}.F \wedge \bigwedge_{\psi \in \mathtt{V}} \bigwedge_{\sigma \in \Sigma} (\neg G_\psi)\sigma$$ satisfiable ?

Fixpoint computation: a challenge

$$\exists \bar{x}.F \wedge \bigwedge_{\psi \in \mathtt{V}} \bigwedge_{\sigma \in \Sigma} (\neg G_\psi)\sigma \qquad \text{satisfiable ?}$$

Suppose $|\mathtt{V}| = 20000$ and $|\Sigma| = 120$ (e.g. $|\bar{x}| = |\bar{y}| = 5$) then

$$|\bigwedge_{\psi \in \mathtt{V}} \bigwedge_{\sigma \in \Sigma} (\neg G_\psi)\sigma| \sim 2.10^6 \text{clauses}$$

Fixpoint computation: optimizations

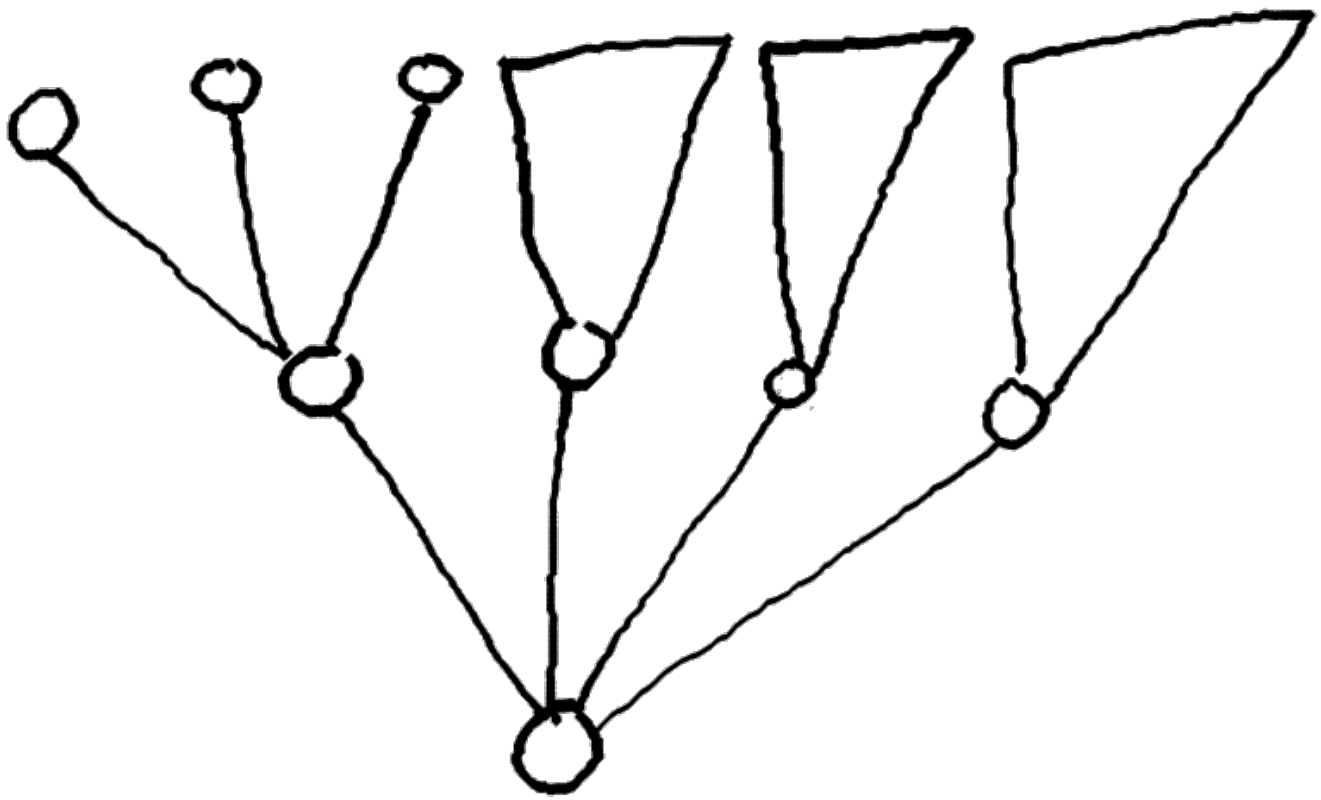
- Fast checks: $G_\psi\sigma \subseteq F$
- Irrelevant permutations:

  $L \in G_\psi\sigma$ and $L' \in F$ and $\neg(L \wedge L')$ is immediate
- A single SMT-context is used for each fixpoint check; it just gets incremented and repeatedly verified

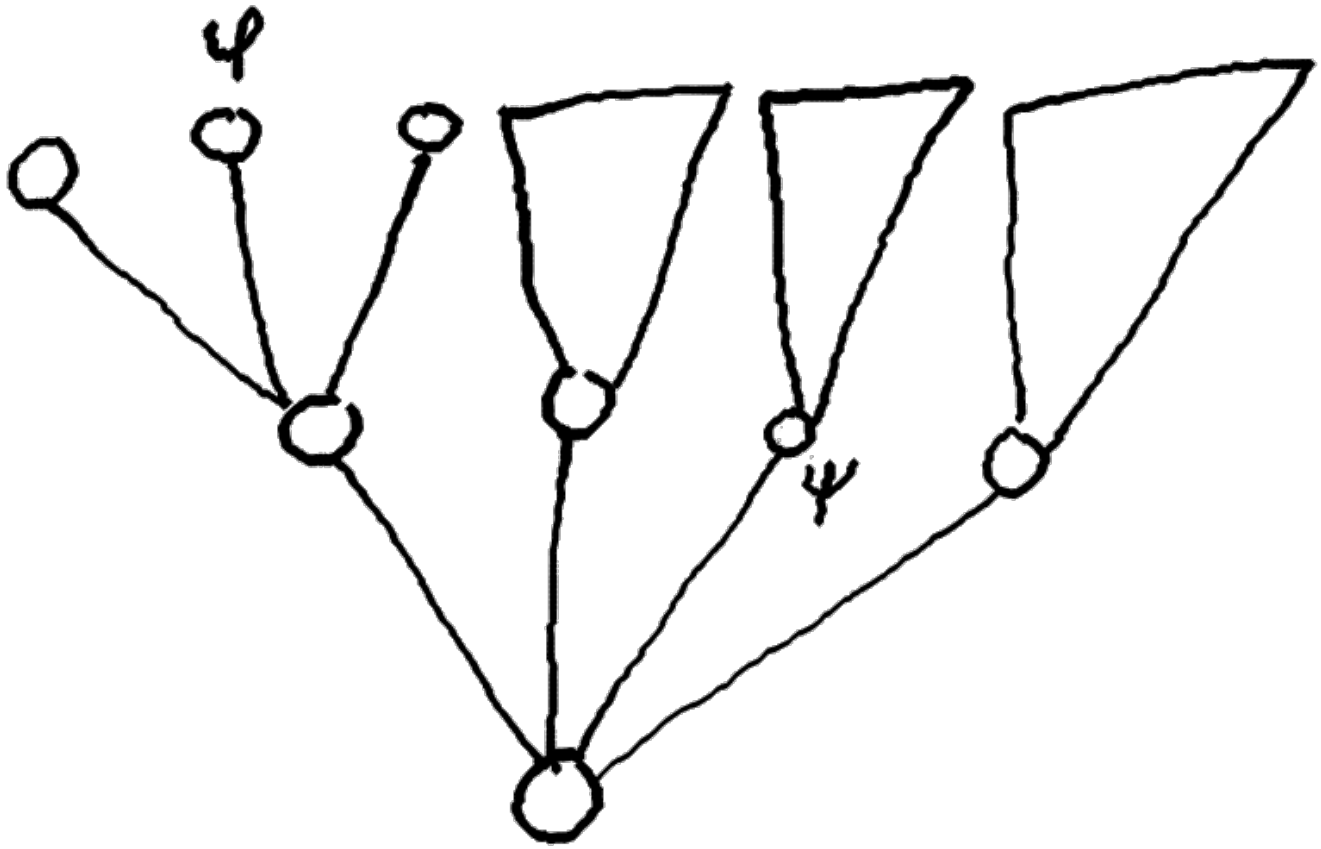
Node deletion

```
V := ∅
push(Q, U)
while not_empty(Q)
  φ := pop(Q)
  if φ ∧ I sat then return(unsafe)
  if ¬(φ ⊨ ⋁_{ψ∈V} ψ) then
    V := V ∪ {φ}
    push(Q, pre(φ))
return(safe)
```

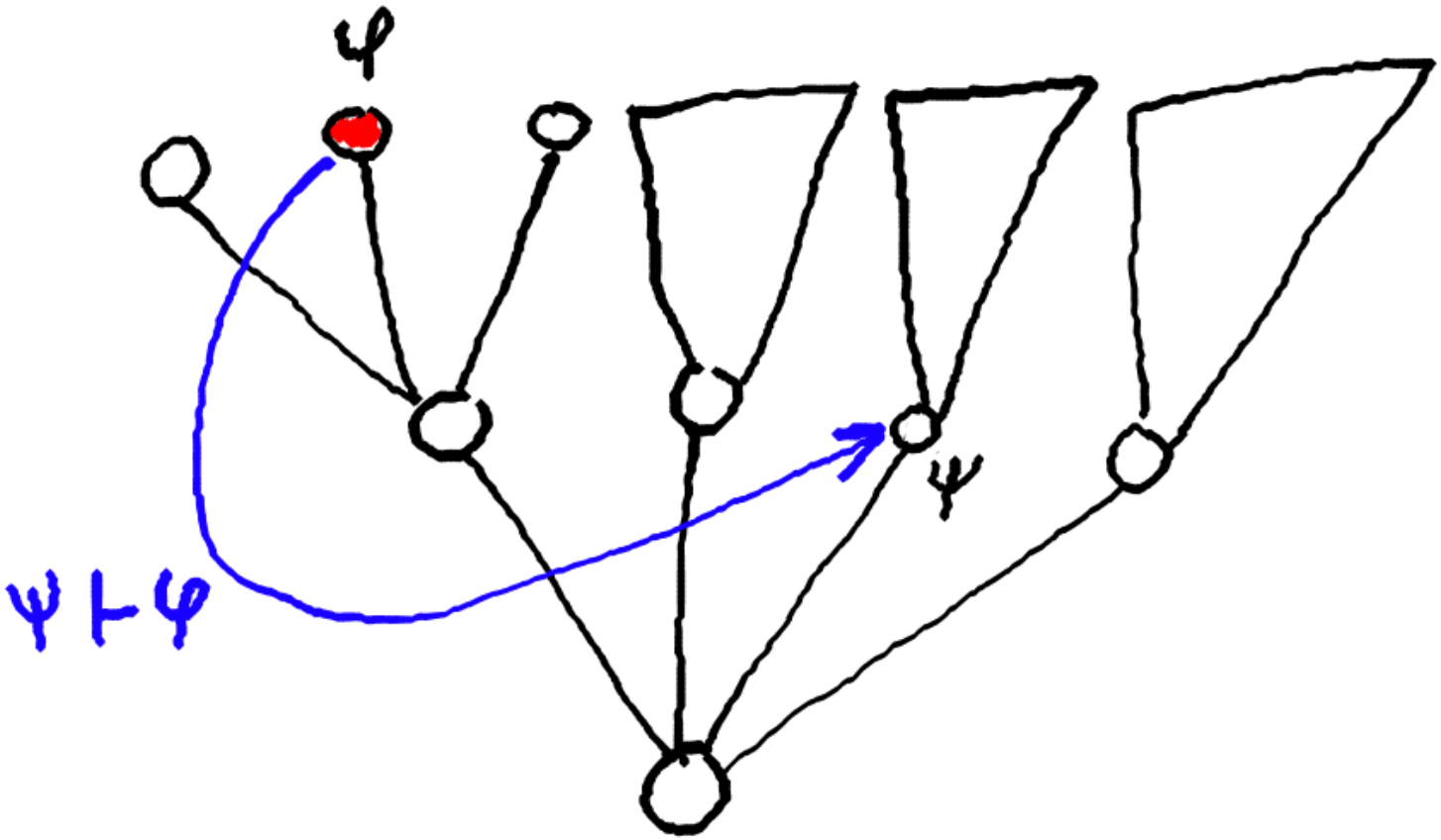
Node deletion : backward simplification



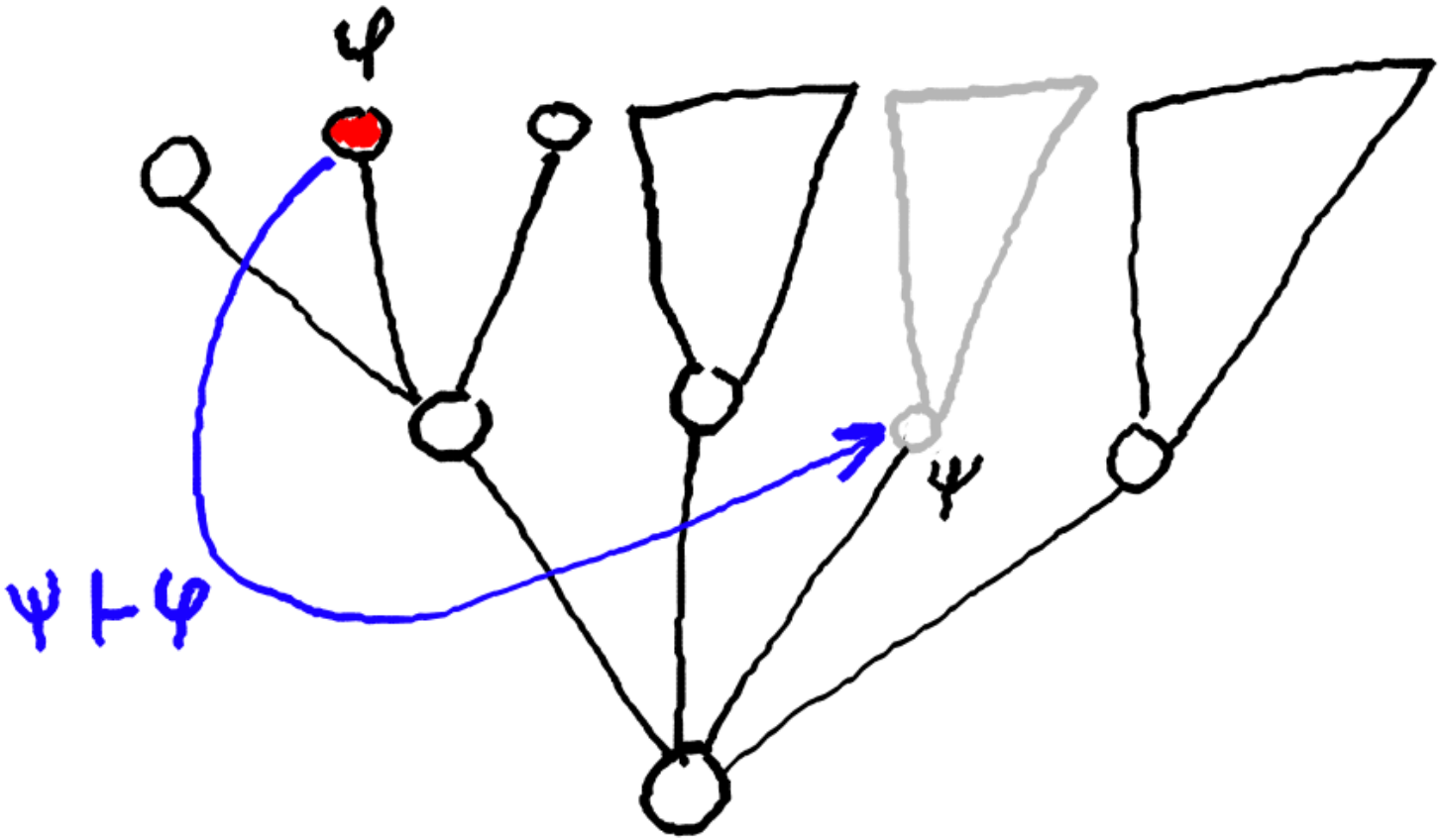
Node deletion : backward simplification



Node deletion : backward simplification



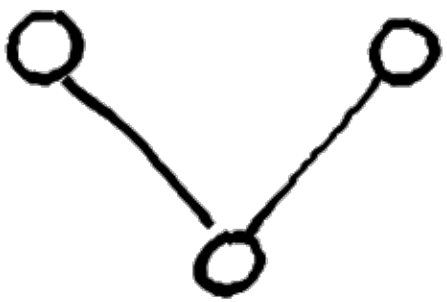
Node deletion : backward simplification



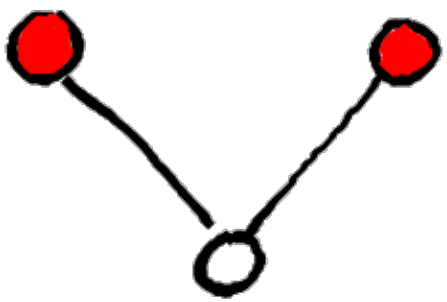
Towards a concurrent implementation

- Based on the Functory Library [Filliâtre, Krishnamani] : master / worker architecture
- Search can be parallelized:
  - Expensive tasks = fixpoint checks
  - Synchronization to keep a precise guidance (BFS)
  - Deletion becomes dangerous

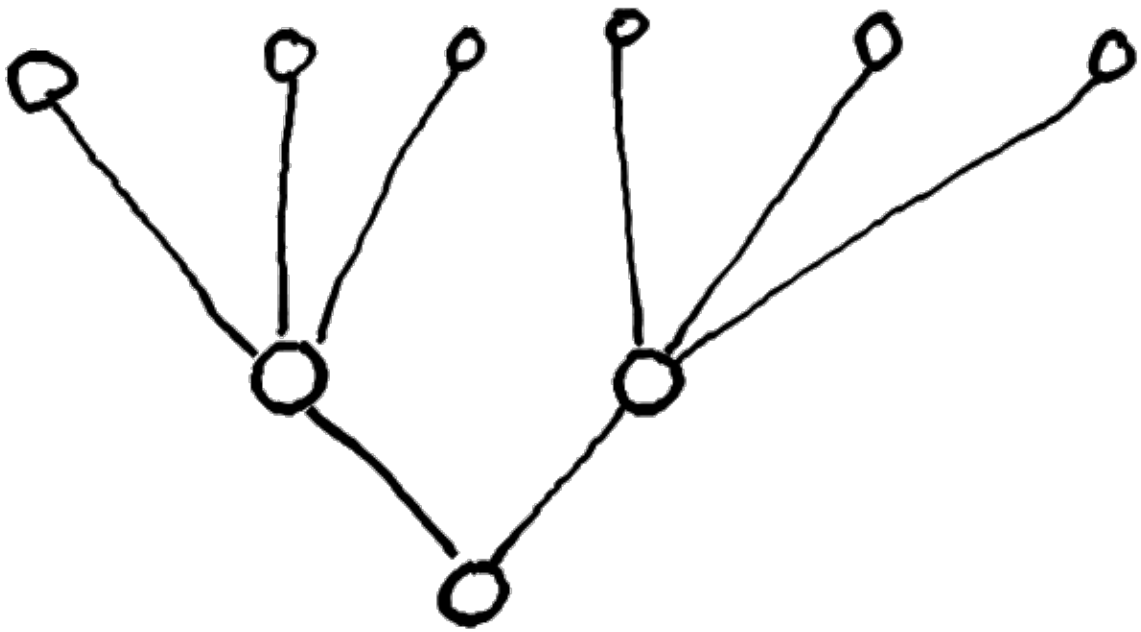
Node deletion in parallel BFS



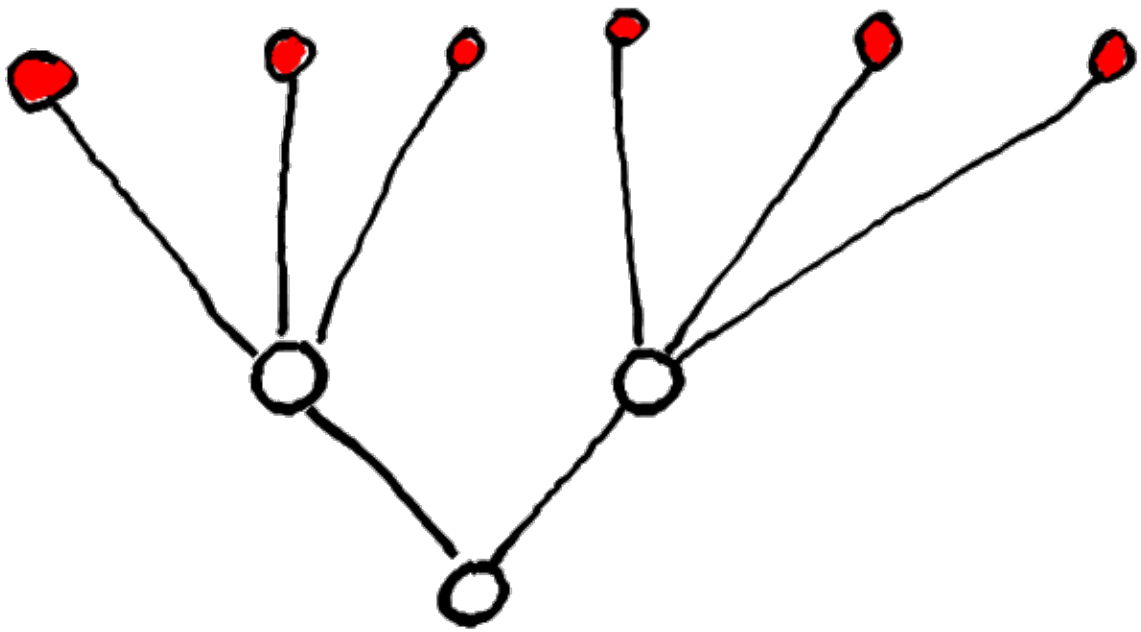
Node deletion in parallel BFS



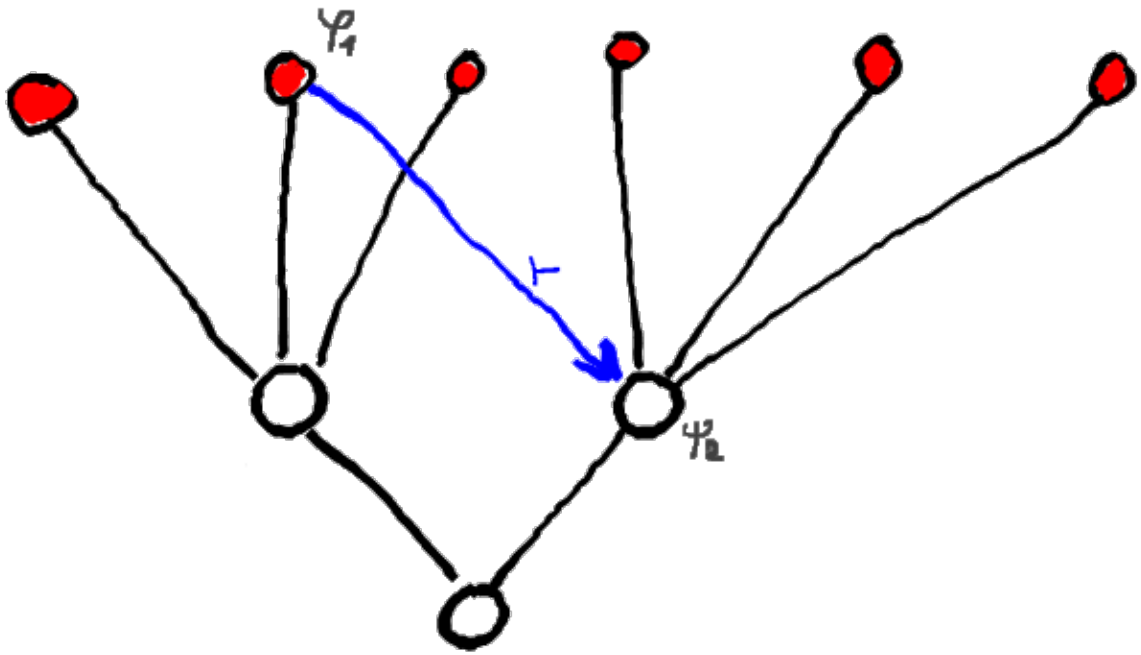
Node deletion in parallel BFS



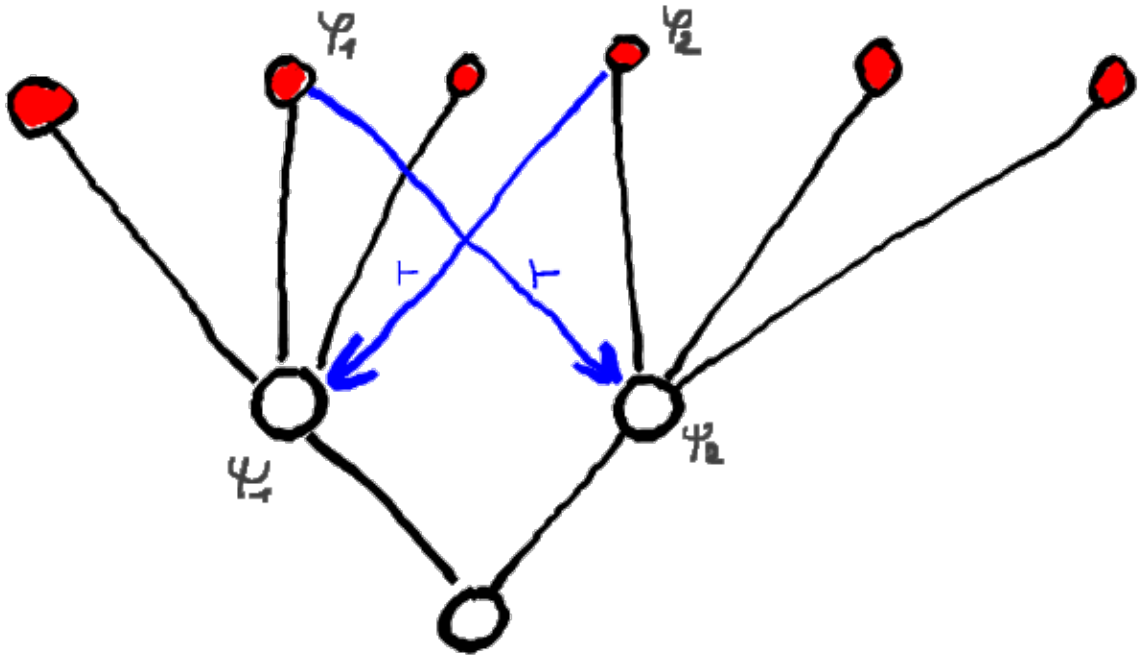
Node deletion in parallel BFS



Node deletion in parallel BFS



Node deletion in parallel BFS



Invariants

How to scale?

Cubicle's benchmarks on academic problems are promising

	Cubicle	CMurphi		
Szymanski_at	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	2h01m	5.68s (4)	2m58s (5)	1h36m (6)

How to scale?

But how to scale up on industrial-like problems?

	Cubicle	CMurphi		
Szymanski_at	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	2h01m	5.68s (4)	2m58s (5)	1h36m (6)
Szymanski_na	T.O.	0.88s (4)	8m25s (6)	7h08m (8)
Flash_nodata	O.M.	4.86s (3)	3m33s (4)	2h46m (5)
Flash	O.M.	1m27s (3)	2h15m (4)	O.M. (5)

O.M. $>$ 20 GB

T.O. $>$ 20 h

What Industrial-Like Means (for us)

A well-known candidate : the FLASH protocol (stanford multiprocessor architecture — 1994)

- Cache-coherence shared memory
- High-performance message passing
- 67 million states for 4 processes ($\sim$40 variables, $\sim$75 transitions)

Our Solution to Scale Up

We designed a new core algorithm for Cubicle that

- infers invariants for parameterized case using finite instances
- inserts and checks them on the fly in a backward reachability loop
- backtracks if necessary

BRAB (Backward Reachability Algorithm with Approximations) [FMCAD 2013]

Example: German-*ish* cache coherence protocol

Client i:

$\texttt{Cache}[i] \in \{\mathsf{E}, \mathsf{S}, \mathsf{I}\}$

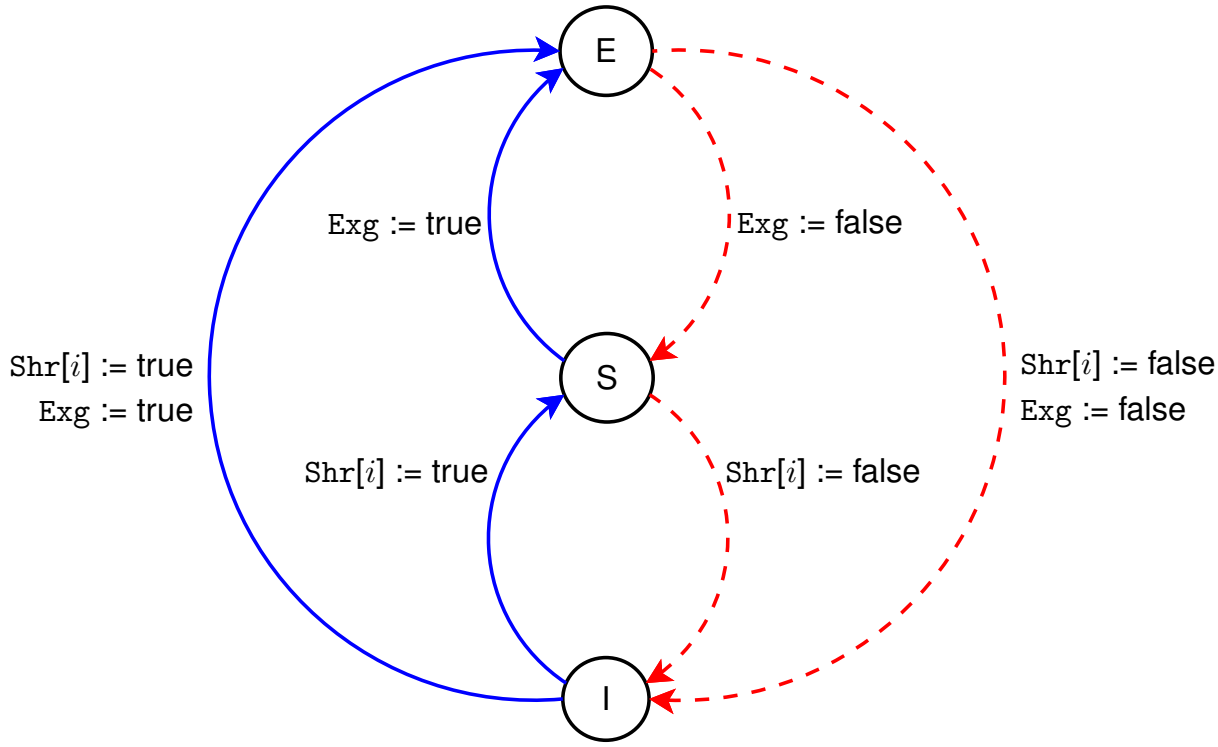
Directory:

$\texttt{Cmd} \in \{\mathsf{rs}, \mathsf{re}, \epsilon\}$

$\texttt{Ptr} \in proc$

$\texttt{Shr}[i] \in \{\mathsf{true}, \mathsf{false}\}$

$\texttt{Exg} \in \{\mathsf{true}, \mathsf{false}\}$



Initial states: $\forall i.\ \texttt{Cache}[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}[i] \ \wedge\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \epsilon$

Unsafe states: $\exists i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$ **?**

(cubes)

Example: German-*ish* cache coherence protocol

Client i:

$\texttt{Cache}[i] \in \{\mathsf{E}, \mathsf{S}, \mathsf{I}\}$

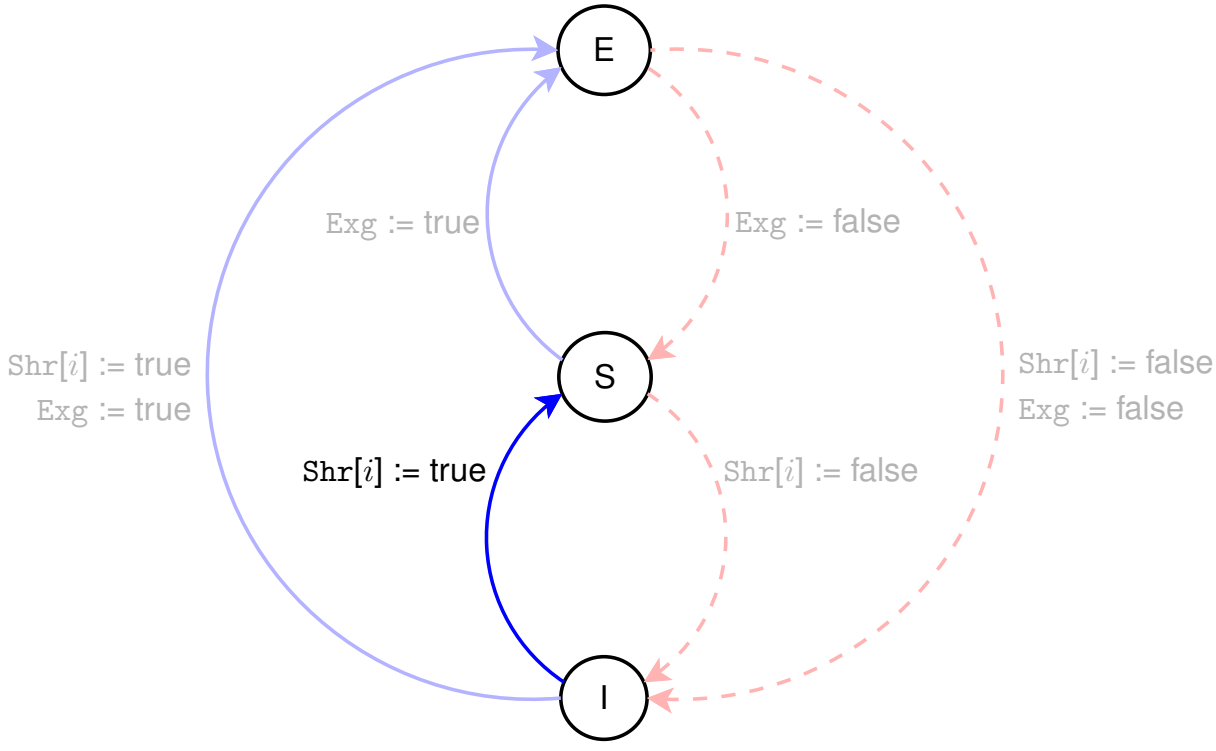
Directory:

$\texttt{Cmd} \in \{\mathsf{rs}, \mathsf{re}, \epsilon\}$

$\texttt{Ptr} \in proc$

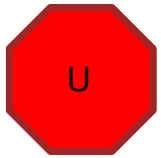
$\texttt{Shr}[i] \in \{\mathsf{true}, \mathsf{false}\}$

$\texttt{Exg} \in \{\mathsf{true}, \mathsf{false}\}$

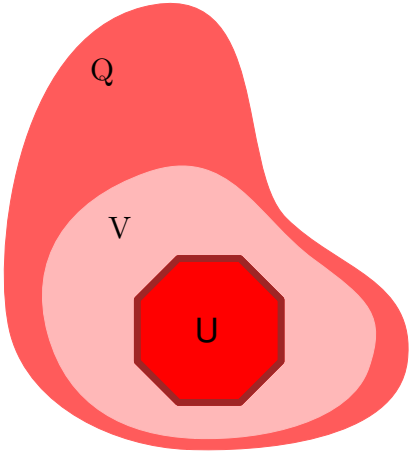


$$t_5 : \exists i. \quad \texttt{Ptr} = i \ \wedge \ \texttt{Cmd} = \mathsf{rs} \ \wedge \ \neg\texttt{Exg} \wedge \\ \texttt{Cmd}' = \epsilon \ \wedge \ \texttt{Shr}'[i] \ \wedge \ \texttt{Cache}'[i] = \mathsf{S}$$

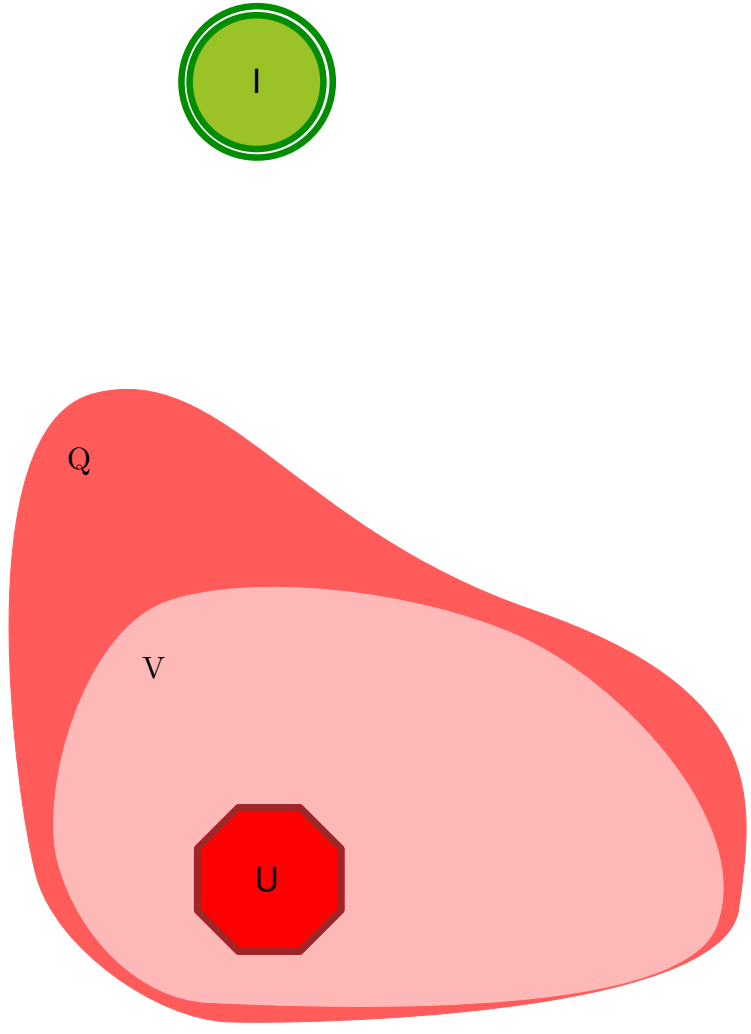
Backward reachability algorithm



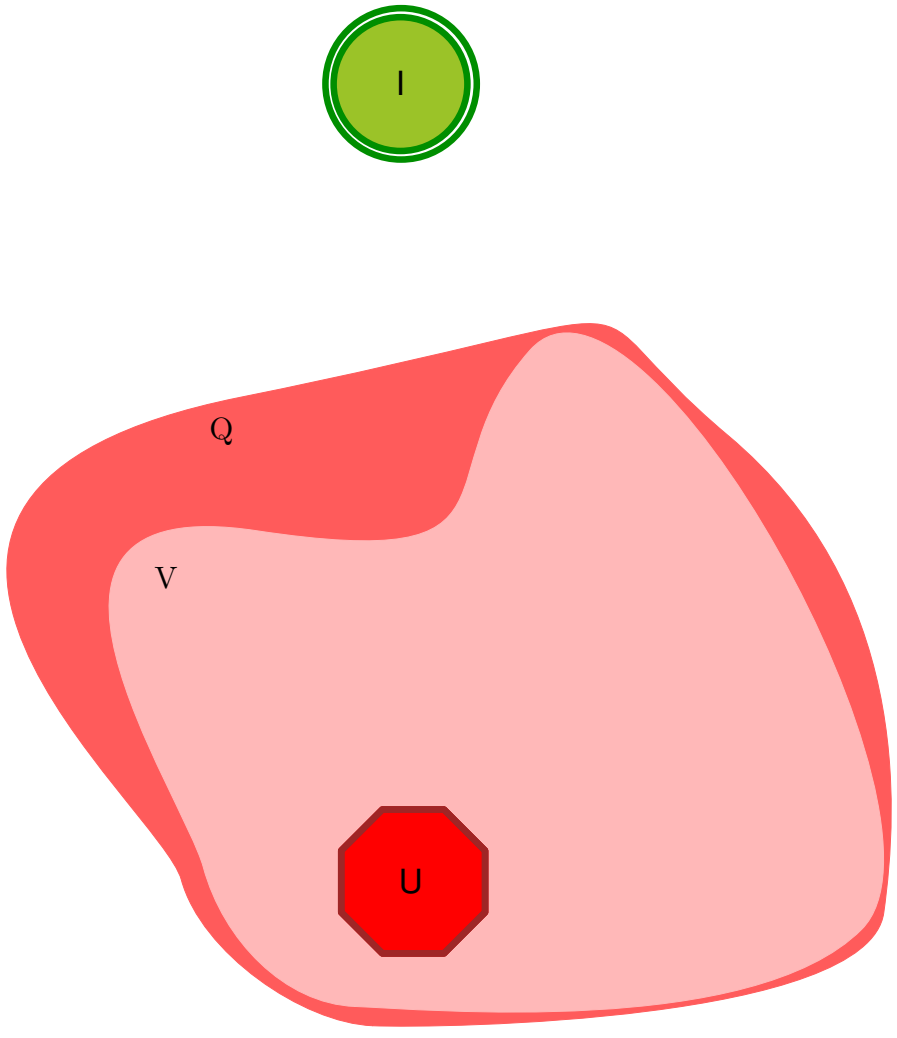
Backward reachability algorithm



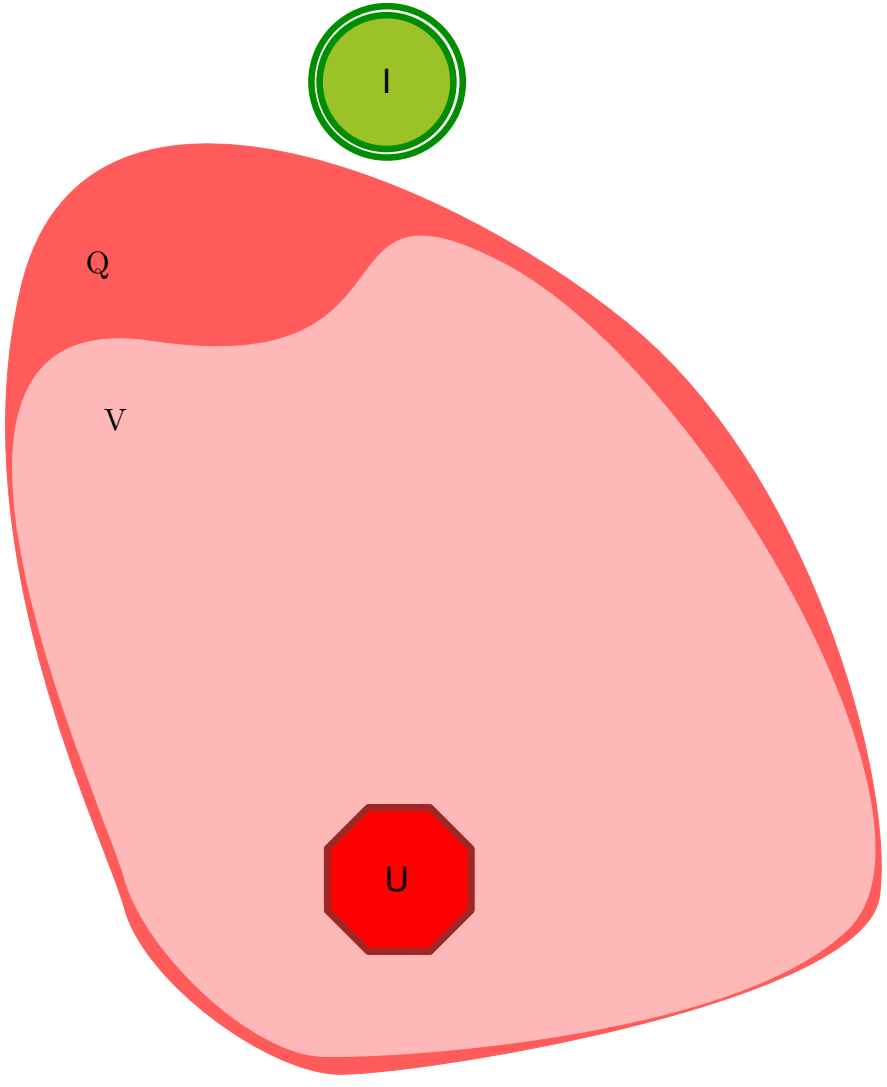
Backward reachability algorithm



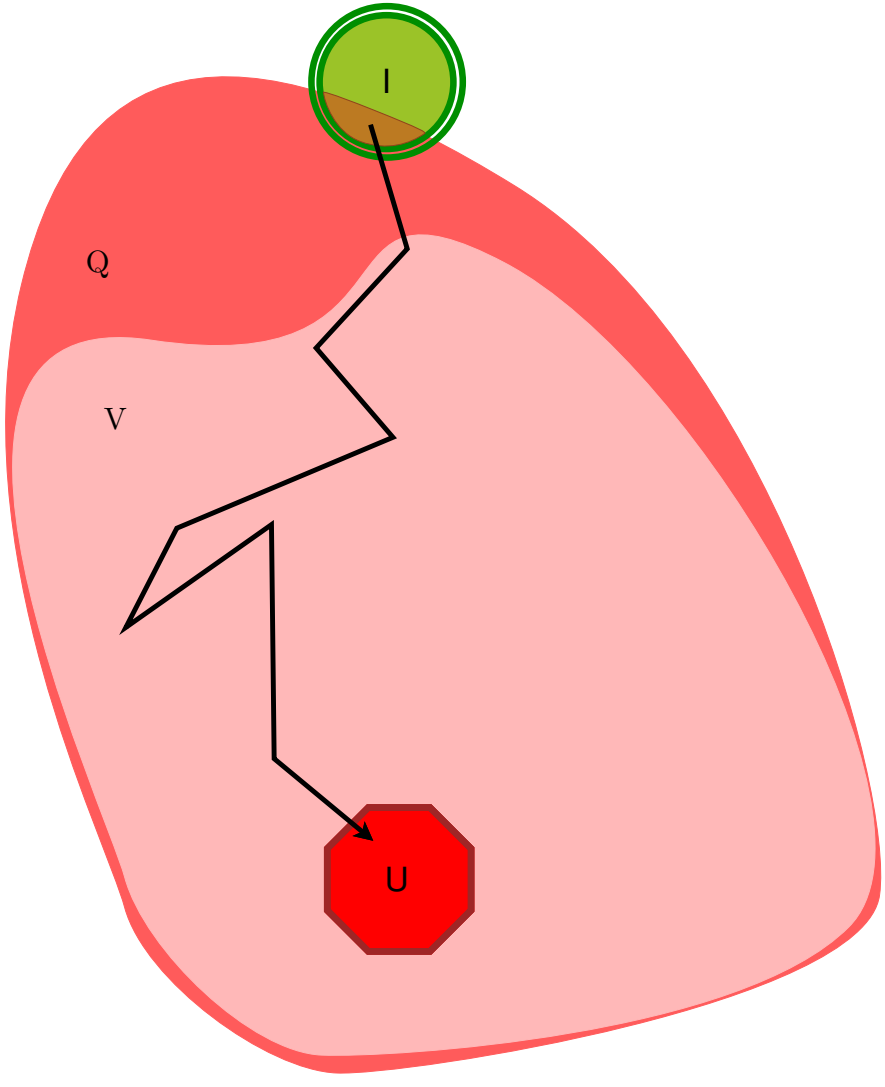
Backward reachability algorithm



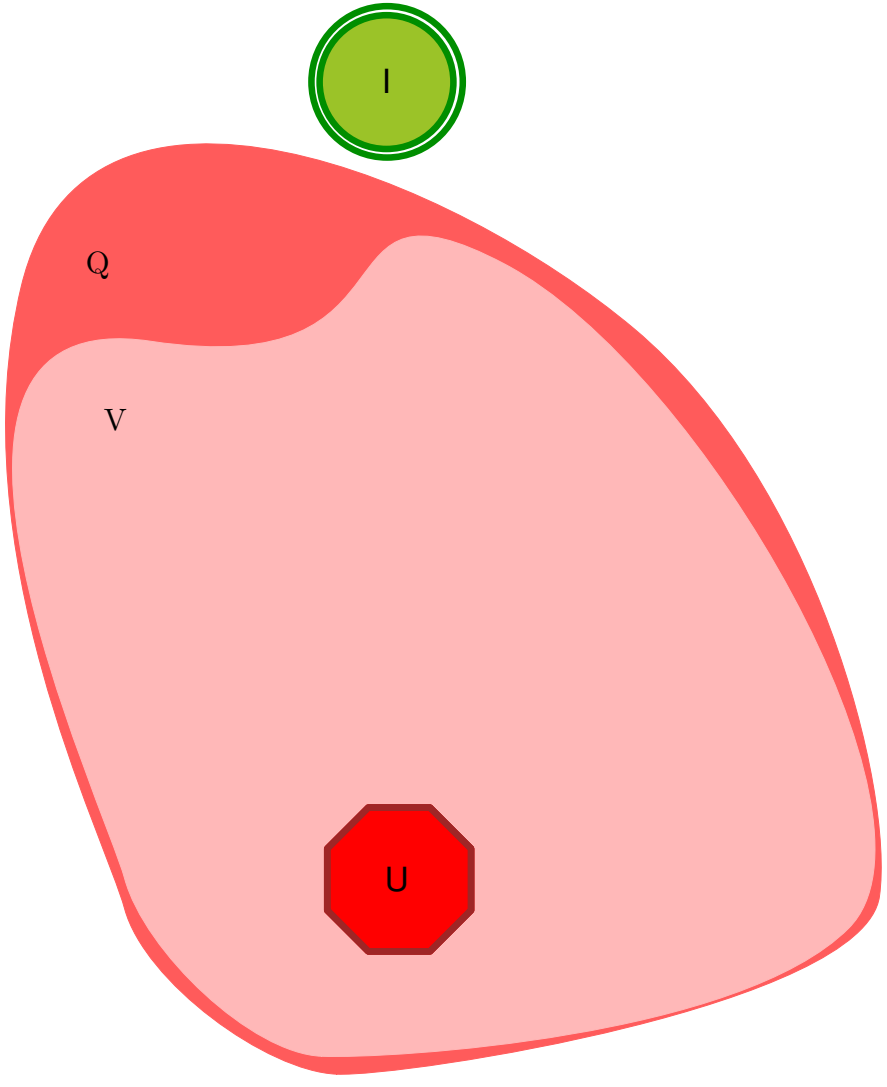
Backward reachability algorithm



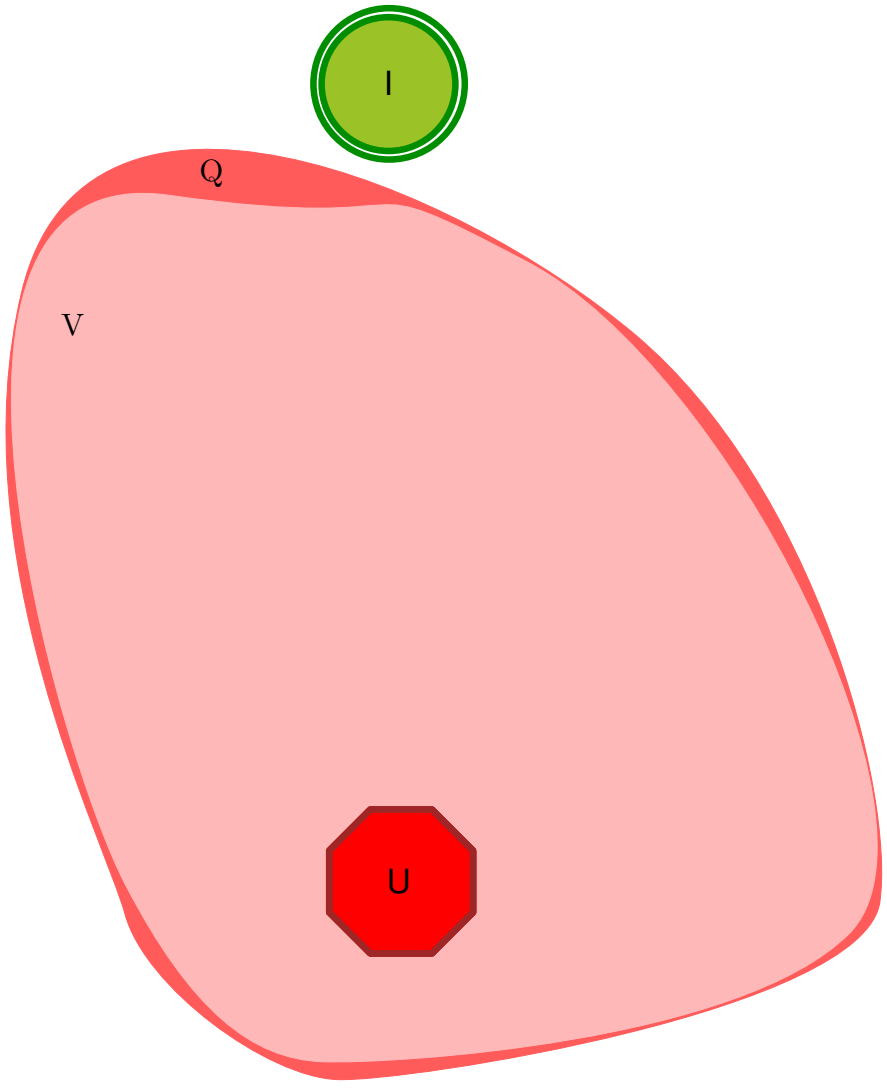
Backward reachability algorithm



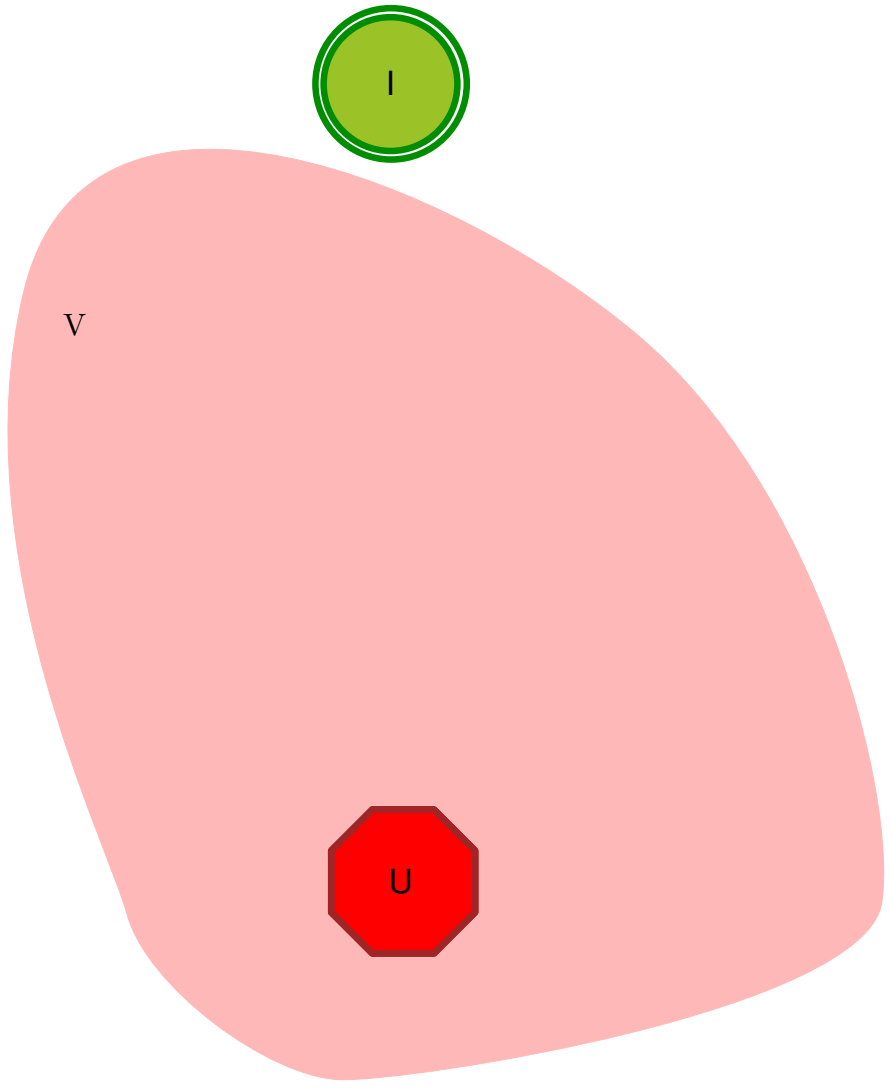
Backward reachability algorithm



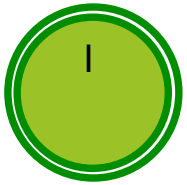
Backward reachability algorithm

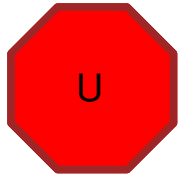


Backward reachability algorithm

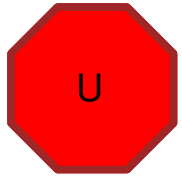


BRAB: intuition

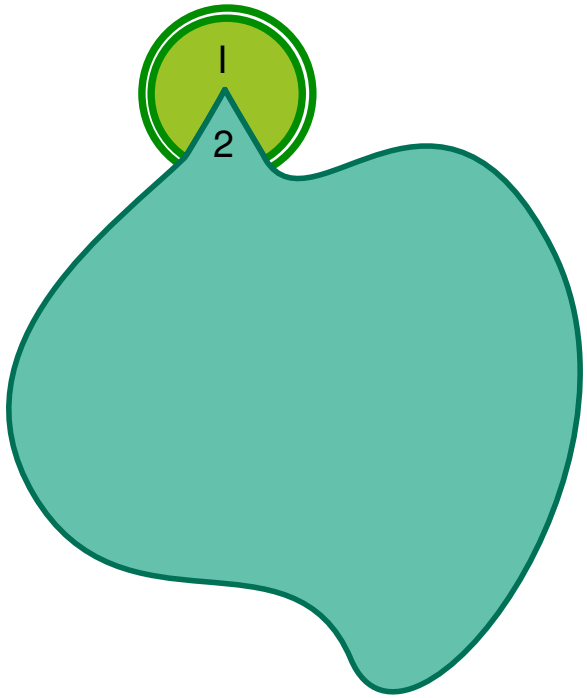


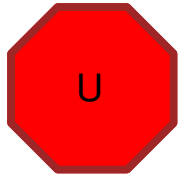


BRAB: intuition

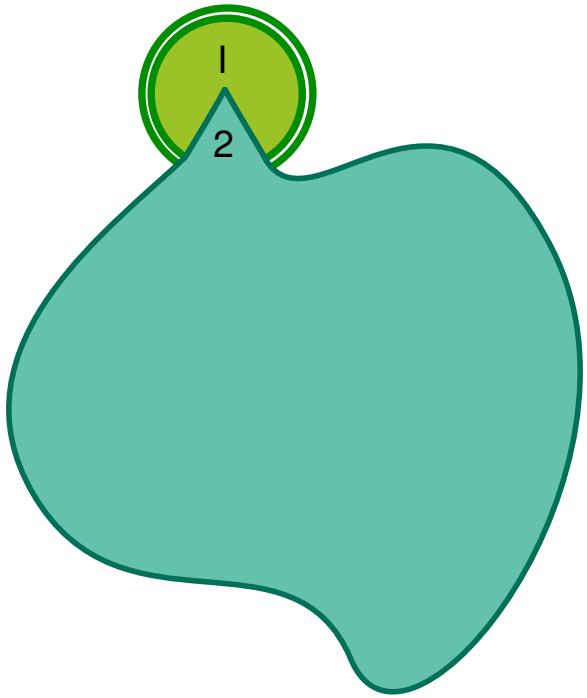


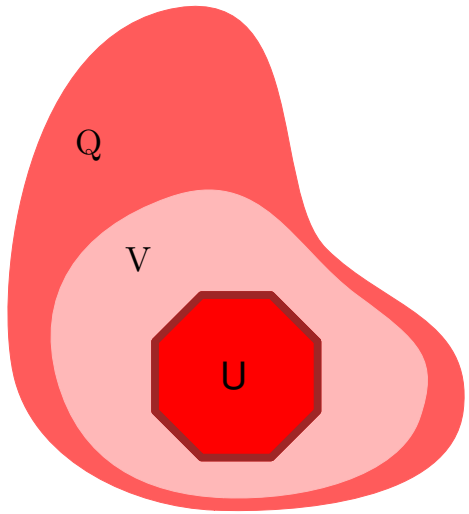
BRAB: intuition



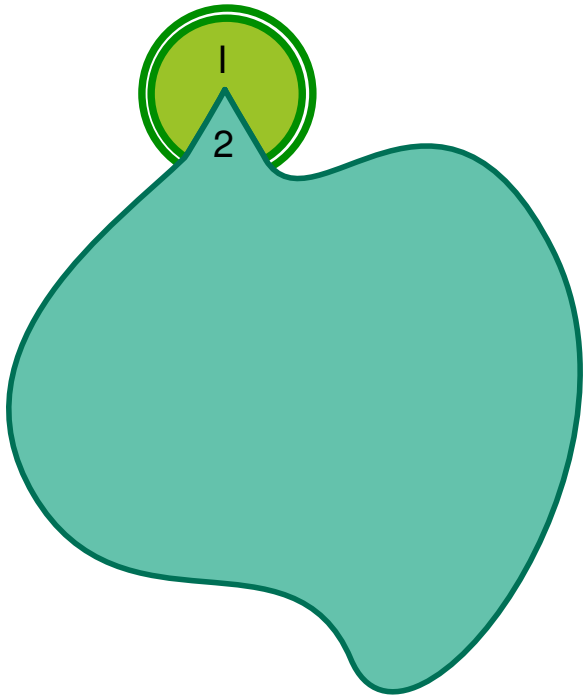


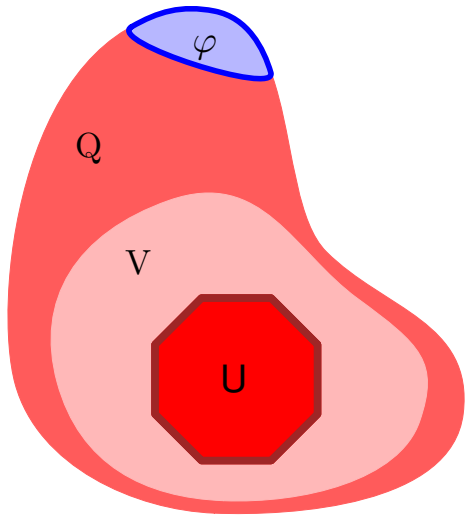
BRAB: intuition



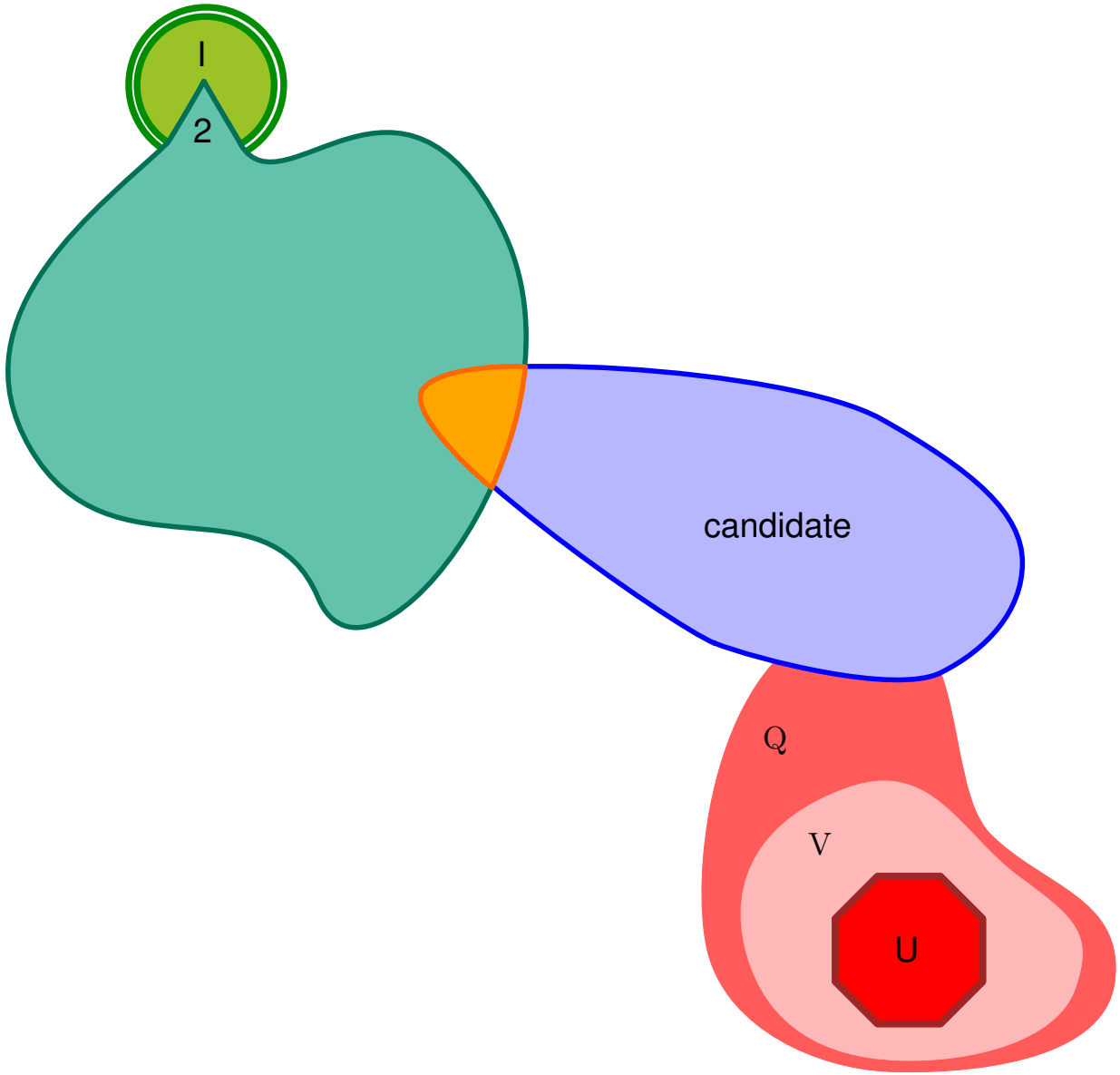


BRAB: intuition

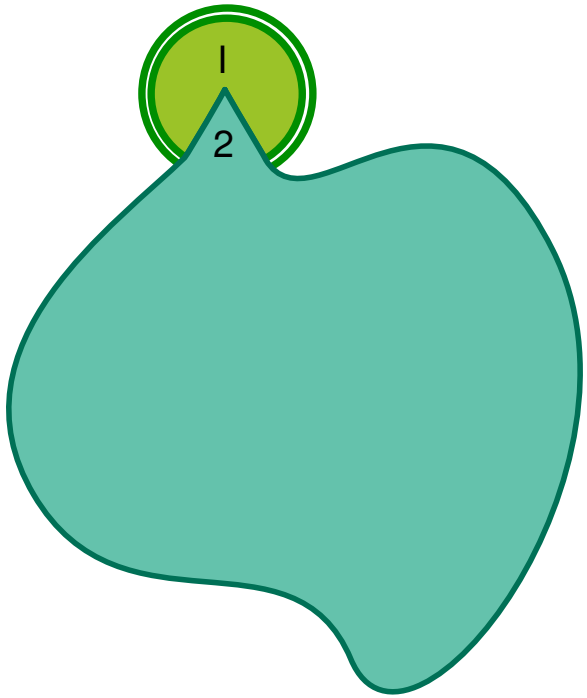


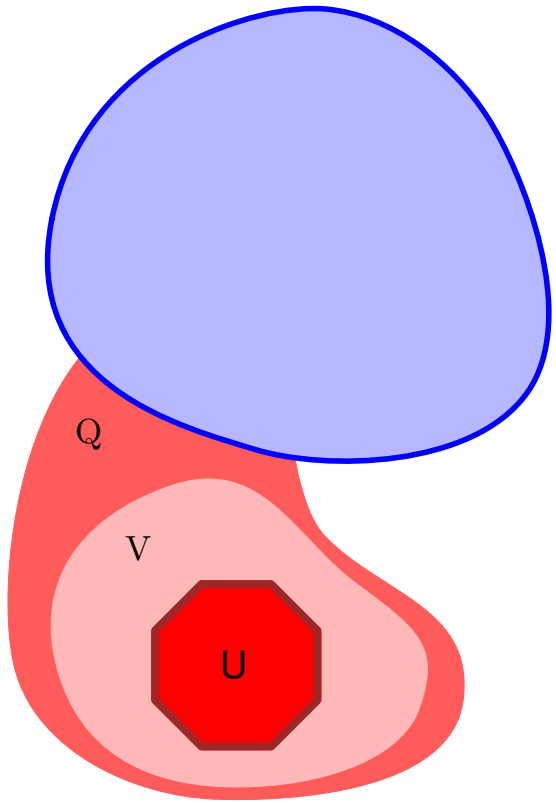


BRAB: intuition

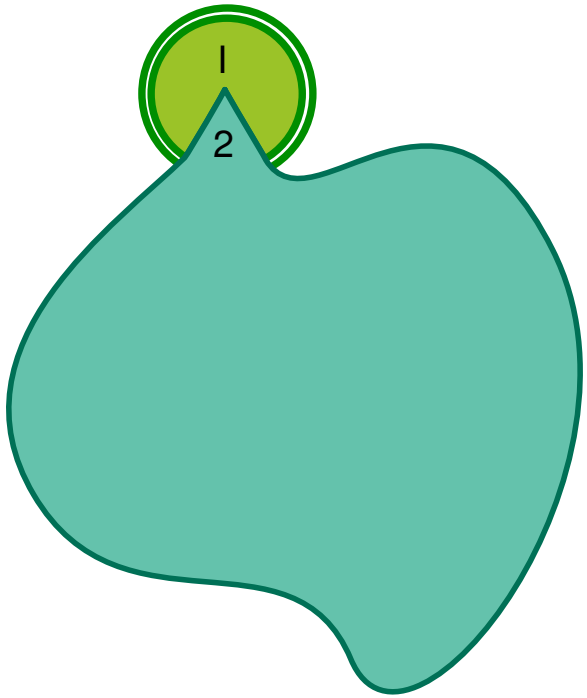


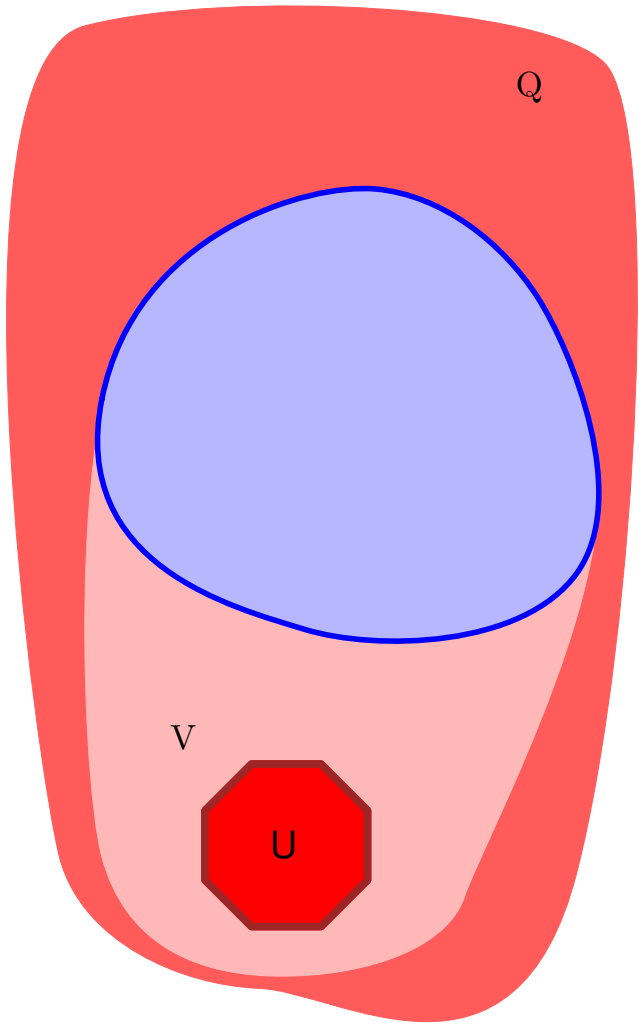
BRAB: intuition



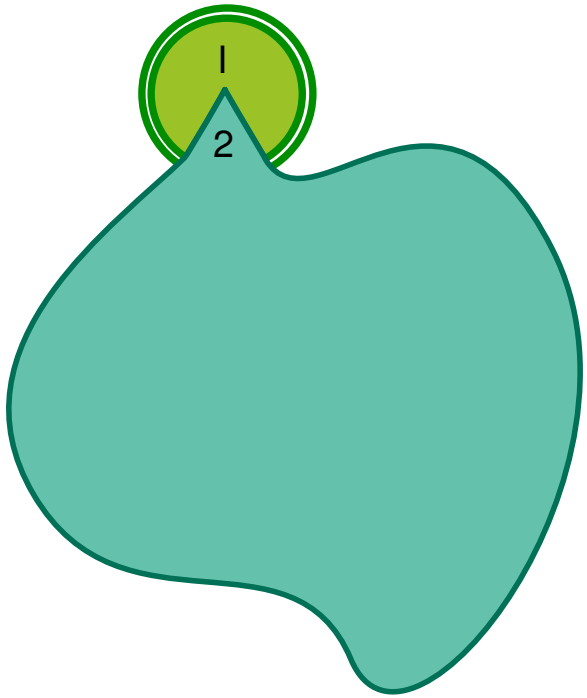


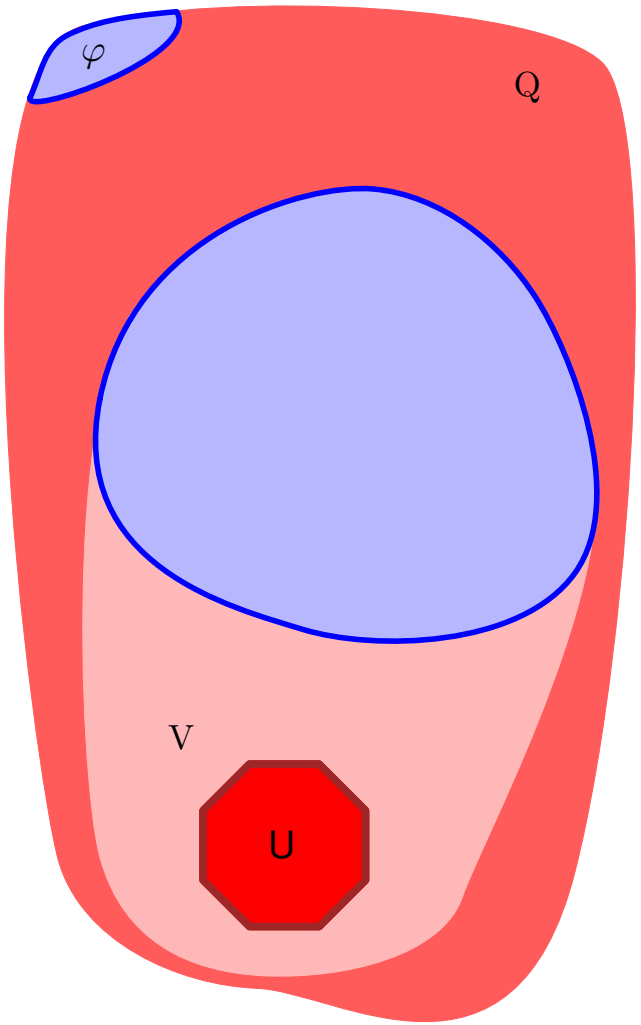
BRAB: intuition



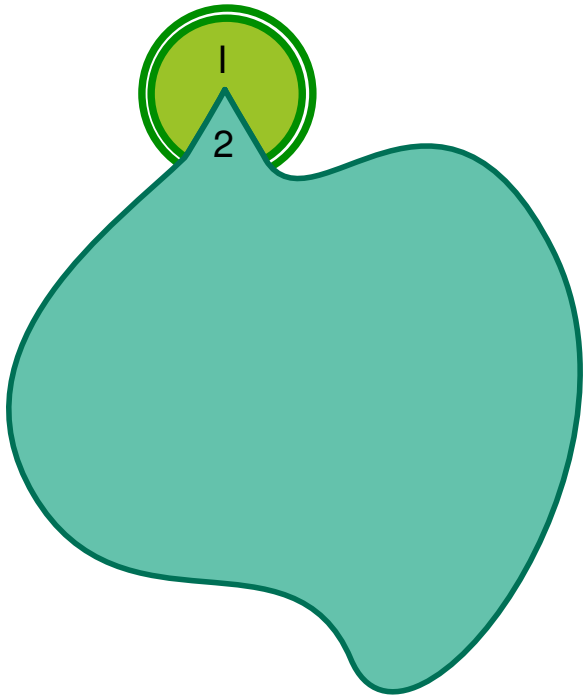


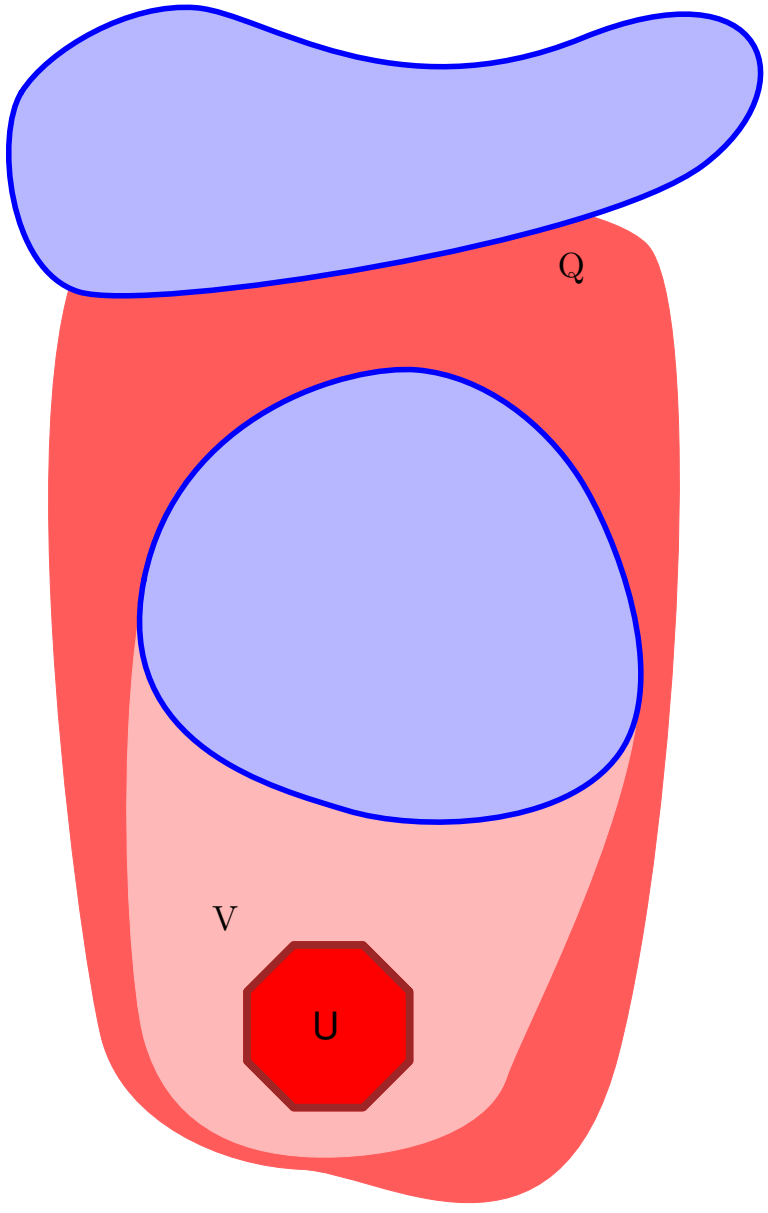
BRAB: intuition



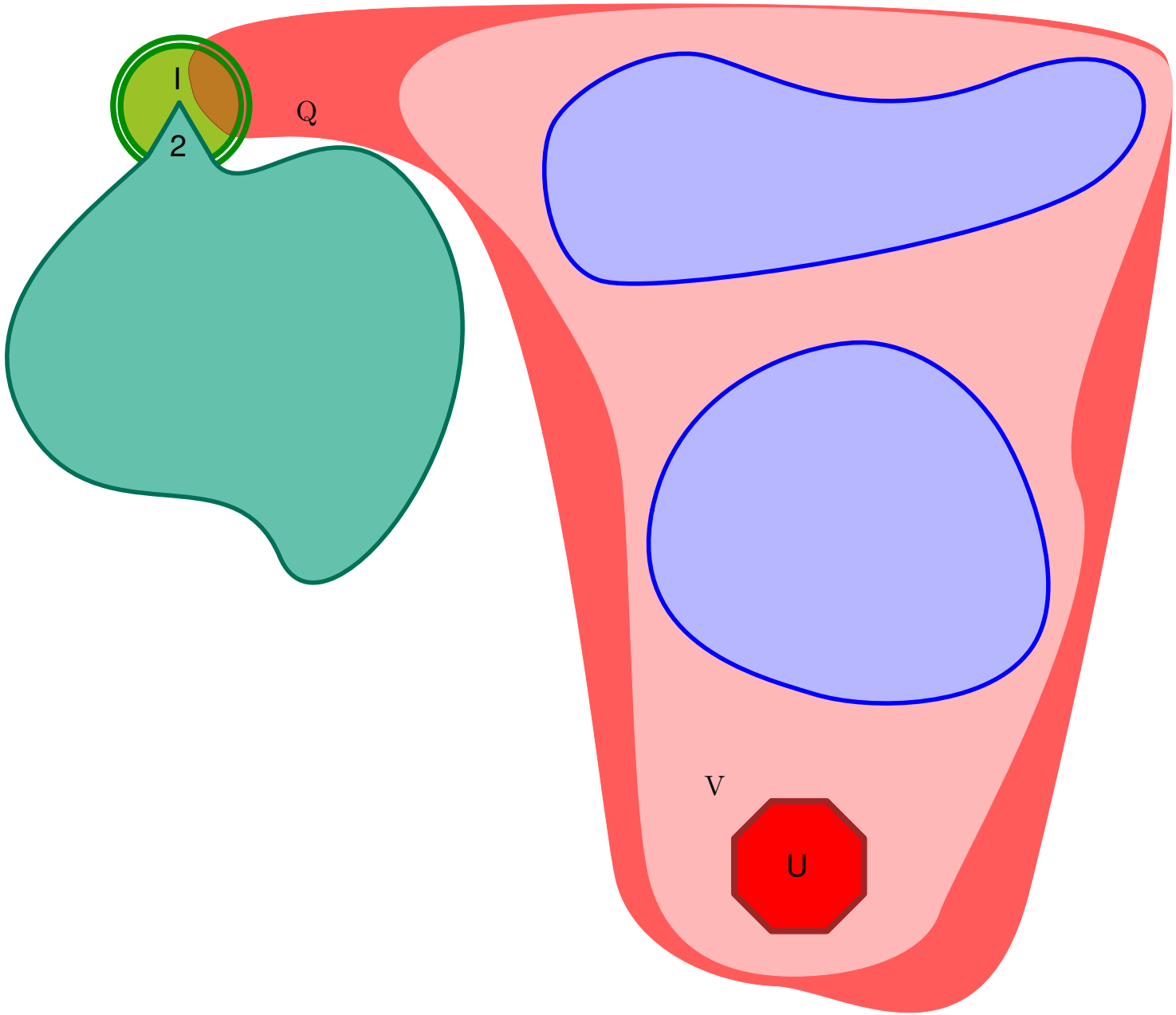


BRAB: intuition

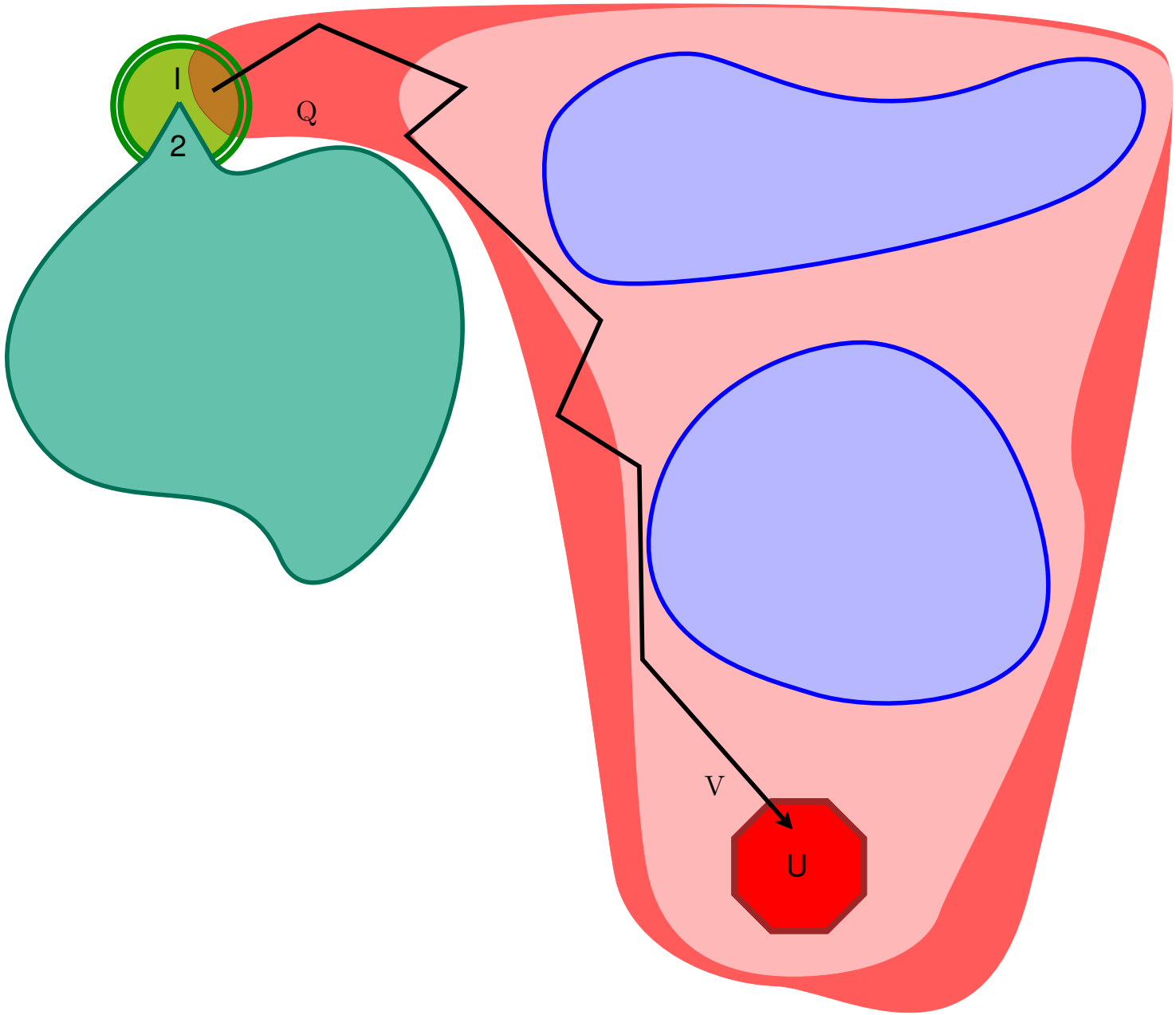




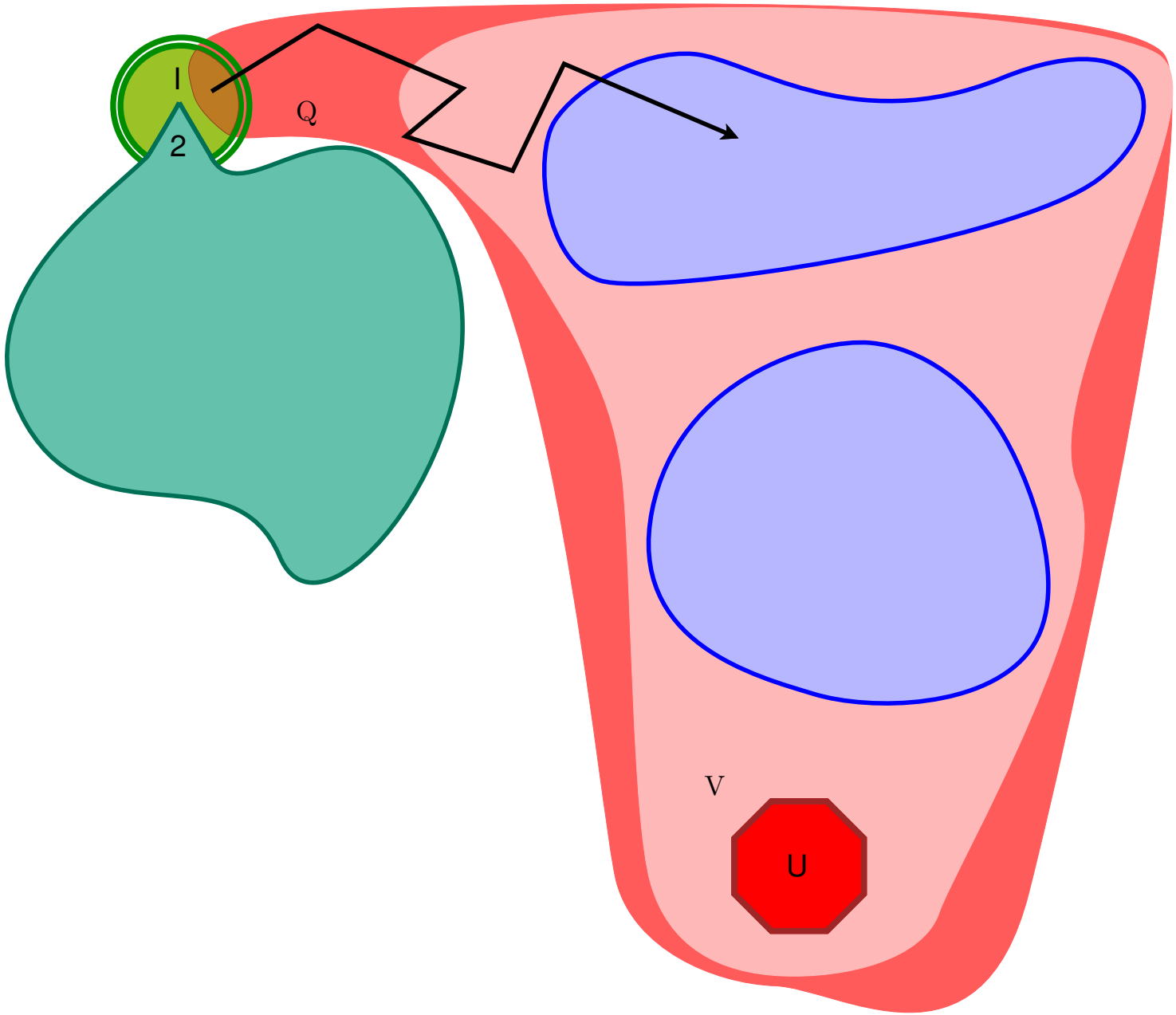
BRAB: intuition



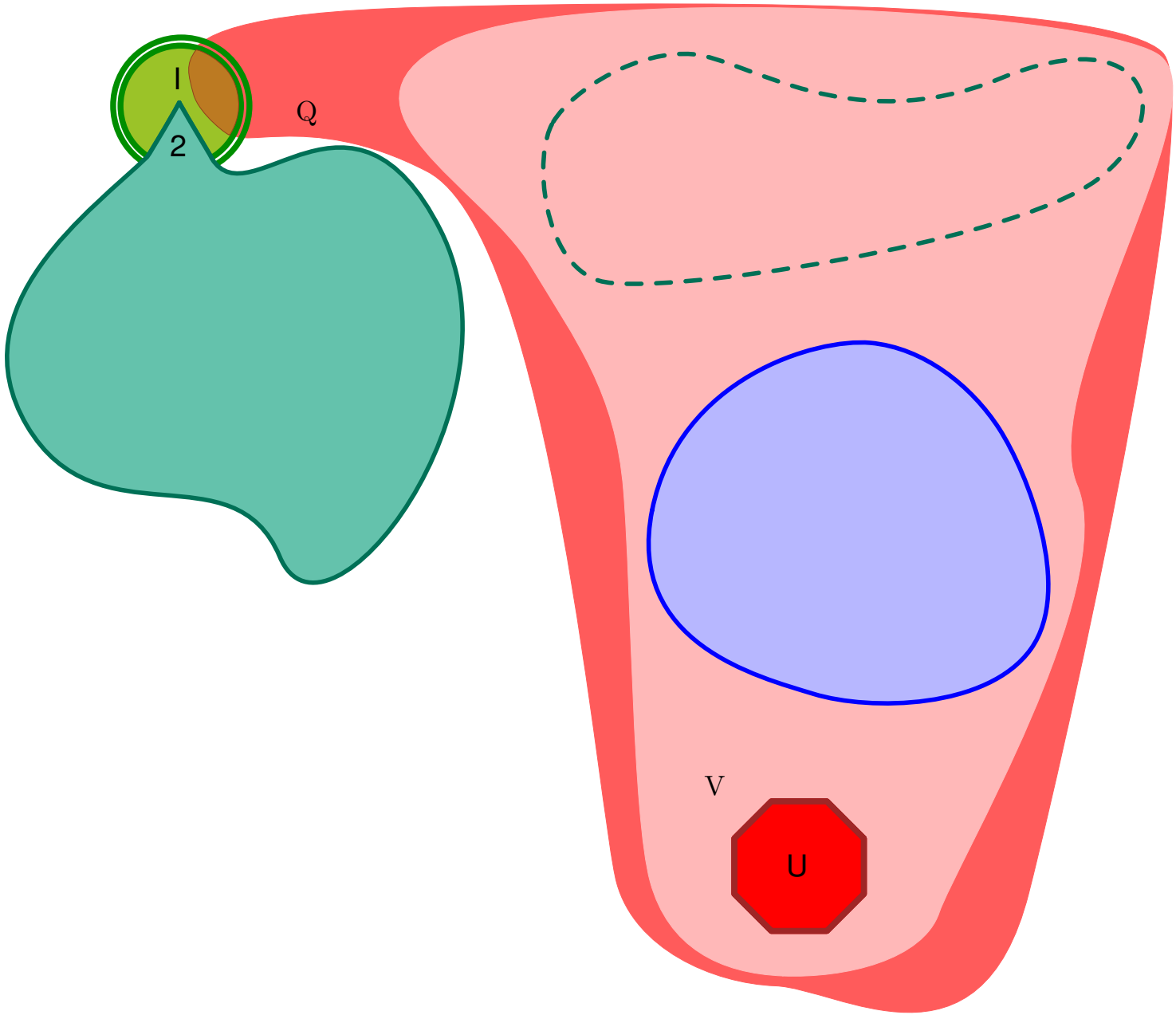
BRAB: intuition



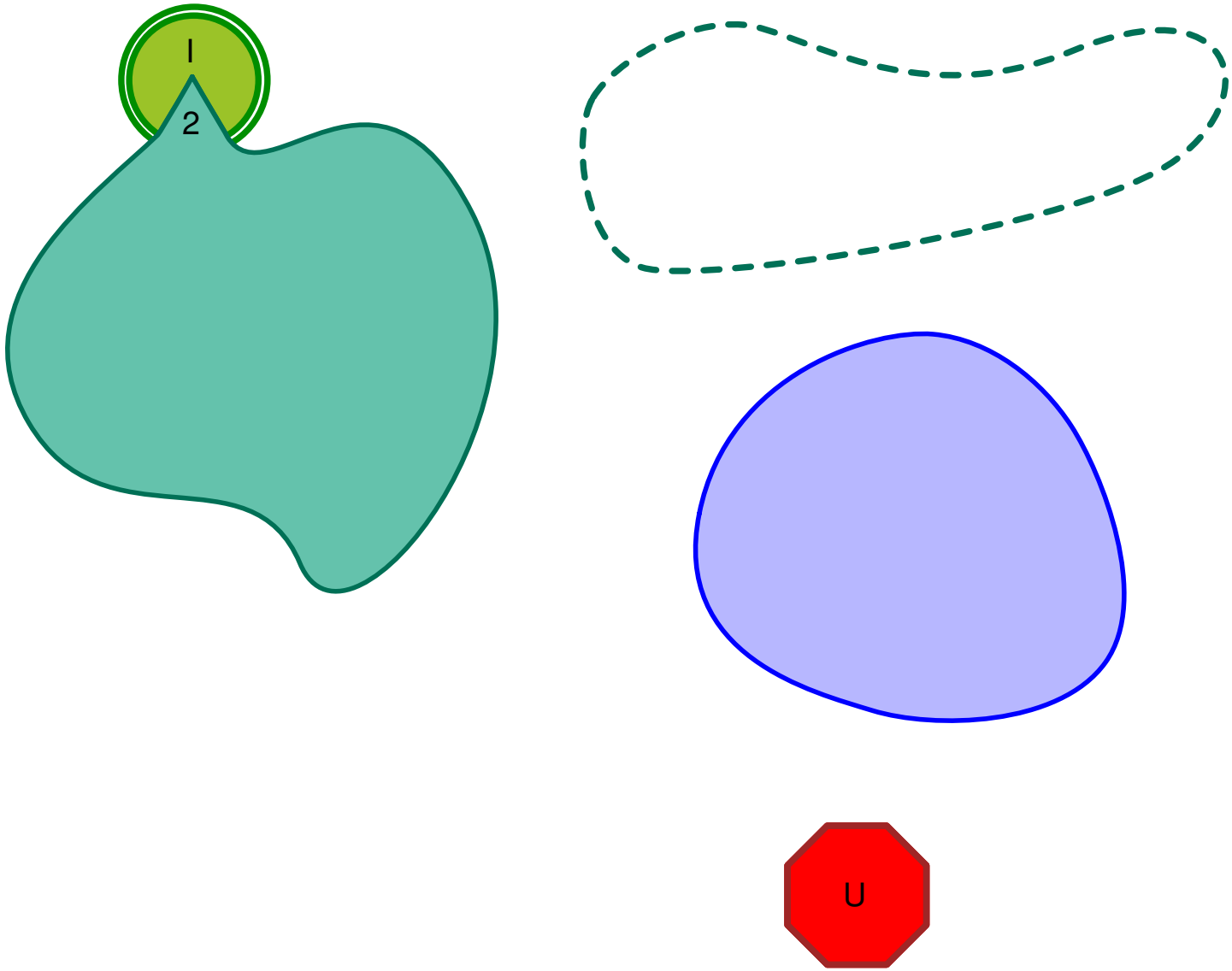
BRAB: intuition



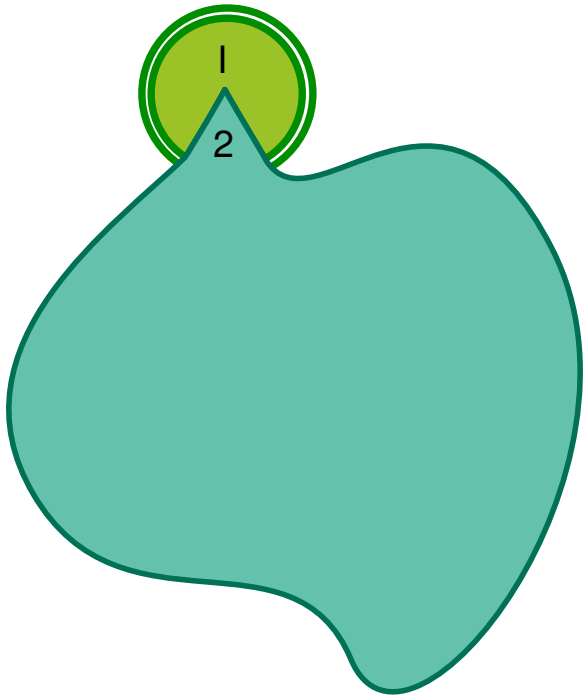
BRAB: intuition

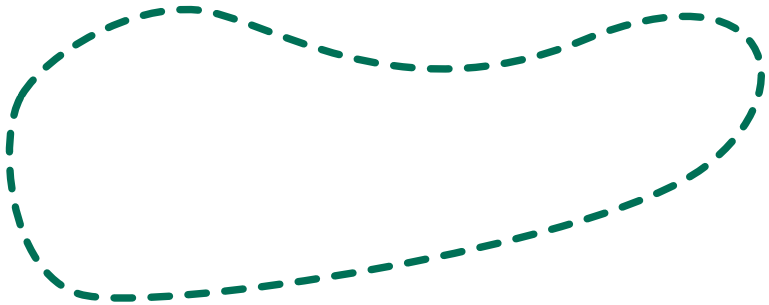


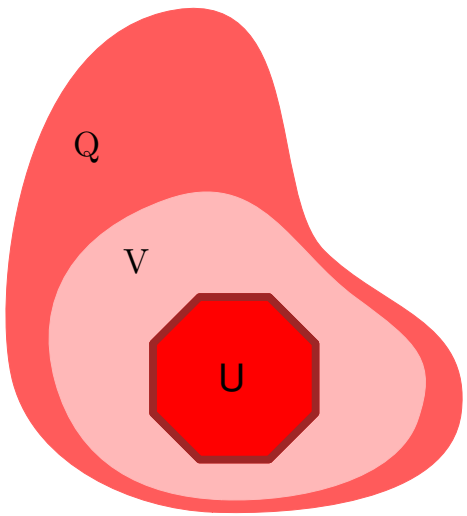
BRAB: intuition



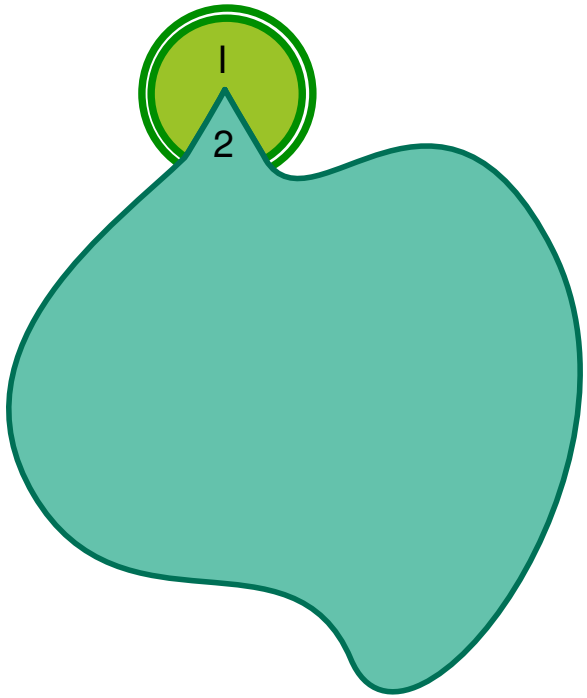
BRAB: intuition

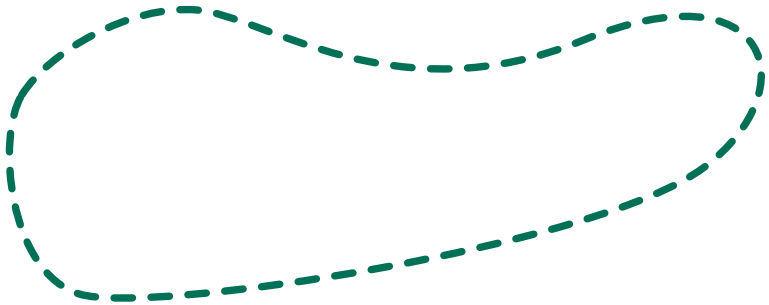


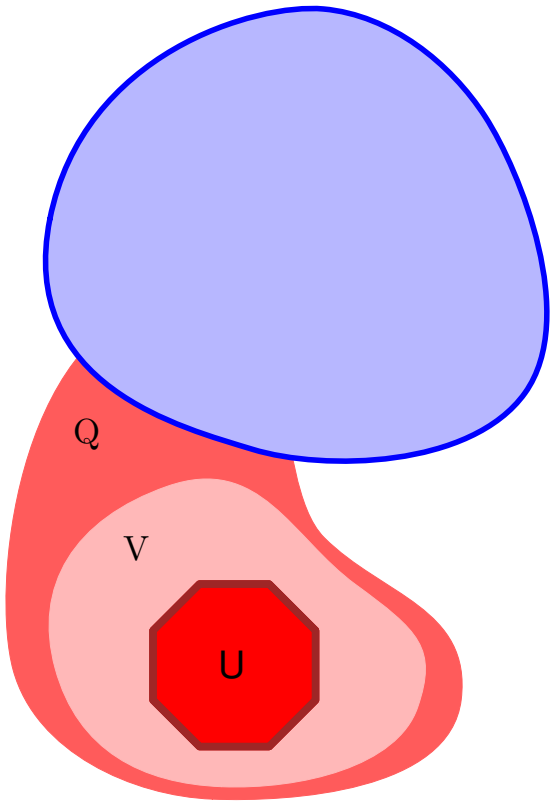




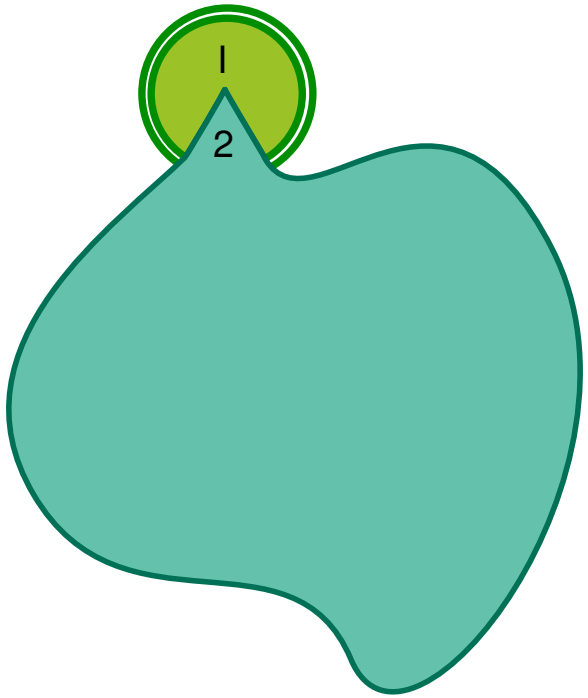
BRAB: intuition

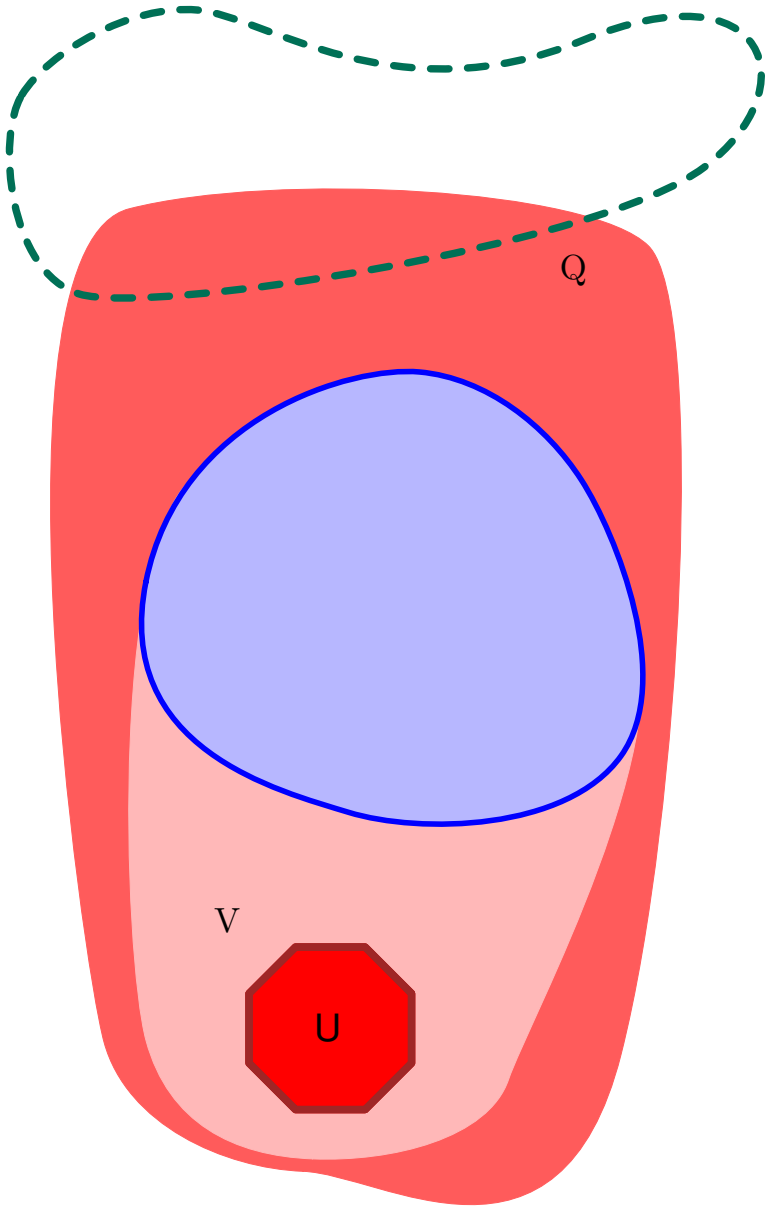




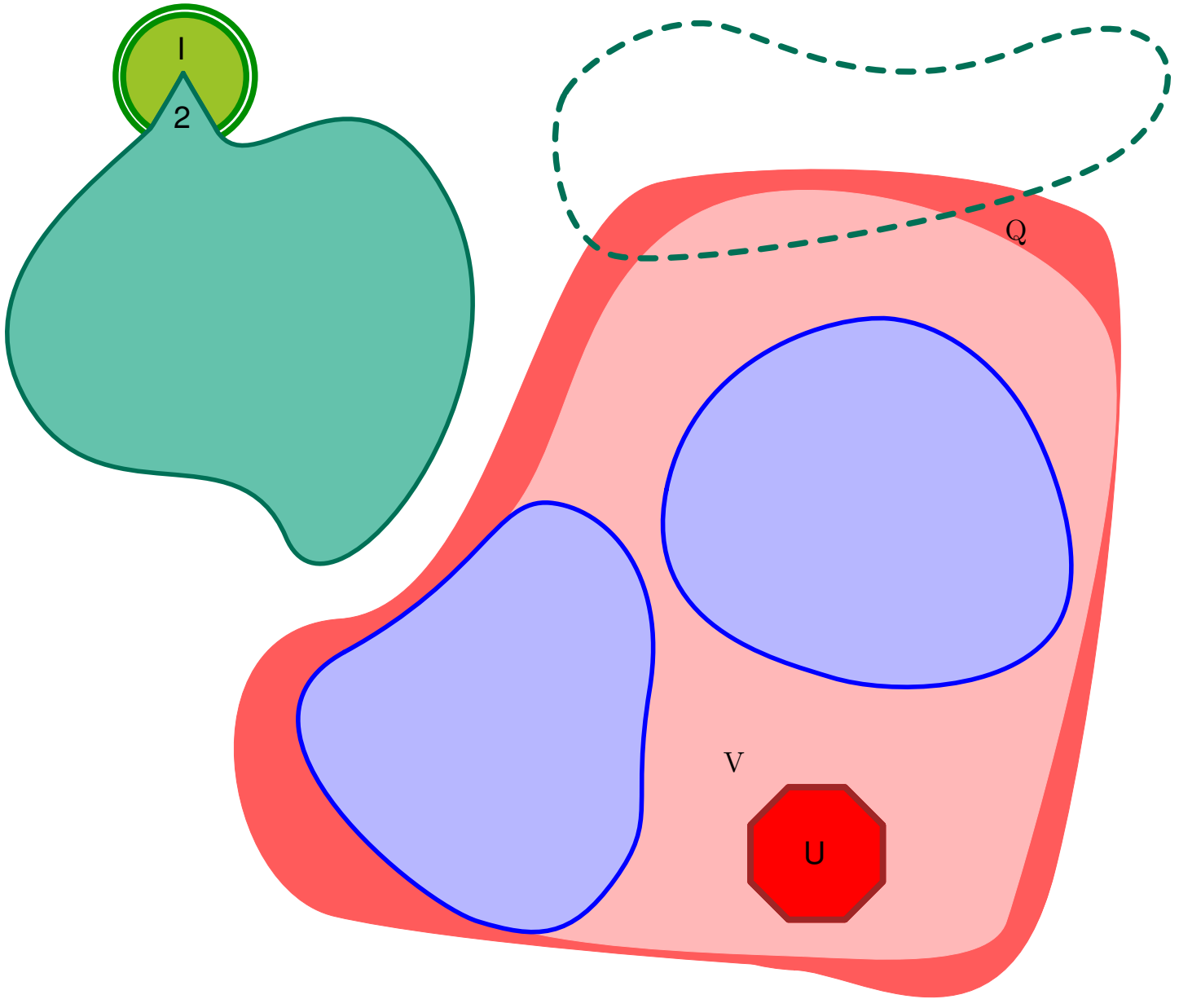


BRAB: intuition

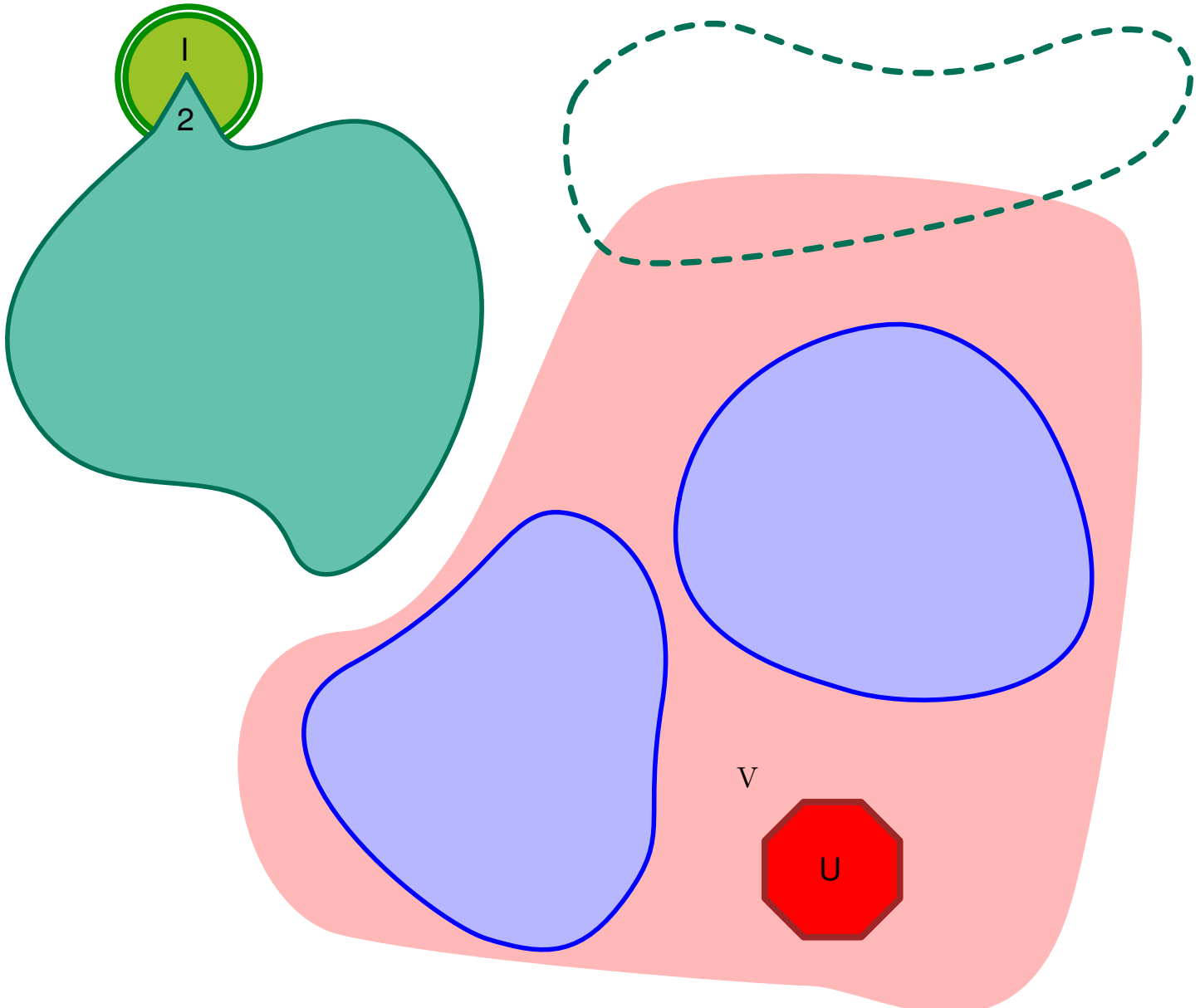




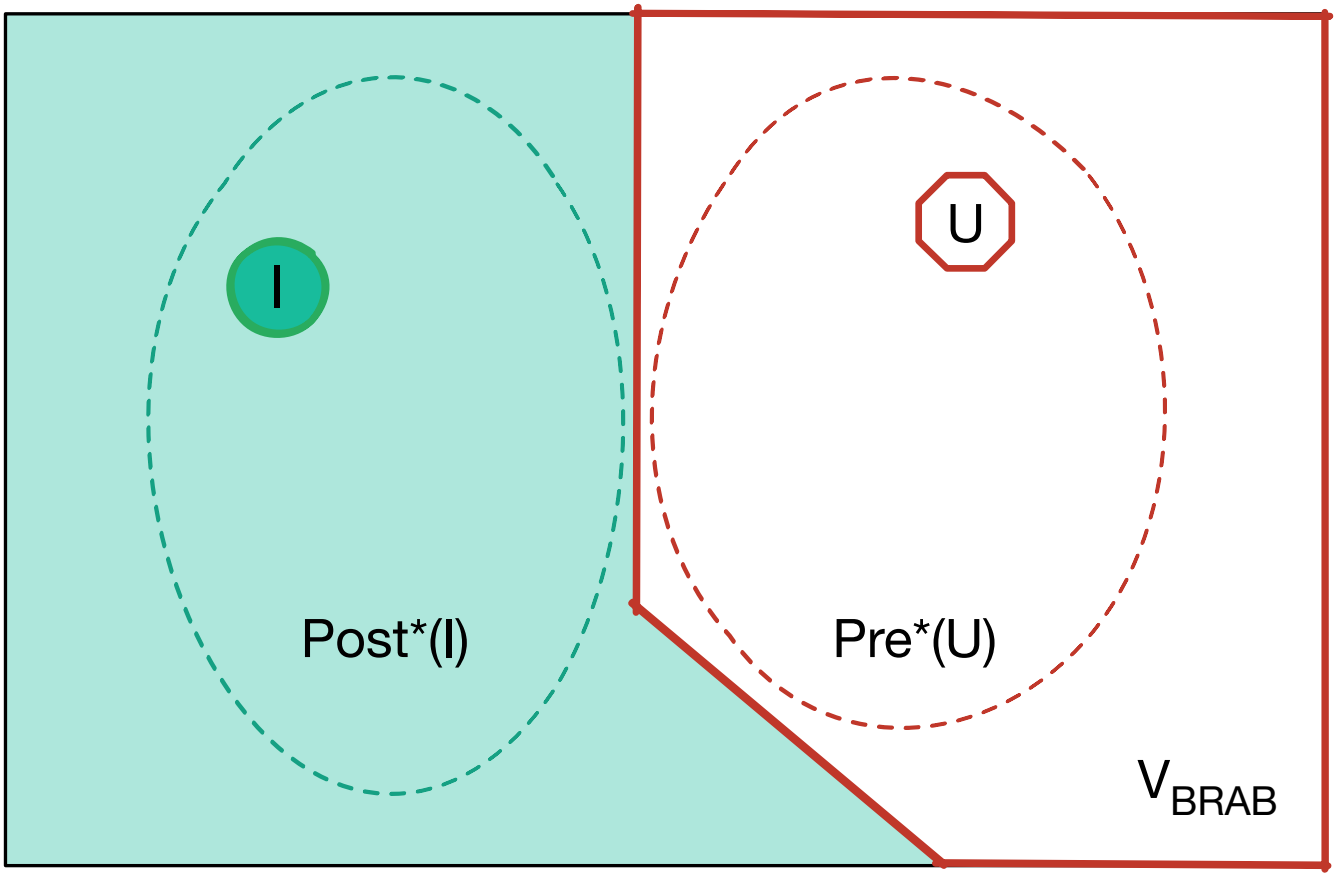
BRAB: intuition



BRAB: intuition



Inductive invariants and BRAB



$$\phi \equiv \neg V_{BRAB}$$

BRAB algorithm

I : inital states U : unsafe states (cubes) $\mathcal{T}$: transitions

---

BRAB ():

B := $\emptyset$; Kind(U) := Orig; From(U) := U;

$\mathcal{M}$:= FWD(d_{max}, k);

while BWDA() = unsafe **do**

  if Kind(F) = Orig **then return** unsafe

  B := B $\cup$ { From(F) };

return safe

BRAB algorithm

I : inital states U : unsafe states (cubes) $\mathcal{T}$: transitions

---

BWD ():

  $\mathtt{V} := \emptyset$;

  push($\mathtt{Q}$, U);

  while not empty($\mathtt{Q}$) **do**

    φ := pop($\mathtt{Q}$);

    if $\varphi \wedge$ I sat **then return** unsafe

    if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ **then**

      $\mathtt{V} := \mathtt{V} \cup \{\varphi\}$;

      push($\mathtt{Q}$, $\mathtt{pre}_{\mathcal{T}}(\varphi)$);

  return safe

BRAB algorithm

I : inital states U : unsafe states (cubes) $\mathcal{T}$: transitions

---

BWDA ():

  $\mathtt{V} := \emptyset$;

  push($\mathtt{Q}$, U);

  while not empty($\mathtt{Q}$) **do**

    φ := pop($\mathtt{Q}$);

    if $\varphi \wedge$ I sat **then return** unsafe

    if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ **then**

      $\mathtt{V} := \mathtt{V} \cup \{\varphi\}$;

      push($\mathtt{Q}$, $\mathrm{Approx}_{\mathcal{T}}(\varphi)$);

  return safe

BRAB algorithm

I : inital states U : unsafe states (cubes) $\mathcal{T}$: transitions

---

Approx$_{\mathcal{T}}$ (φ):

  foreach ψ in `candidates`(φ) **do**

    if $\psi \notin$ B $\wedge$ $\mathcal{M} \not\models \psi$ **then**

      `Kind`(ψ) := Appr;

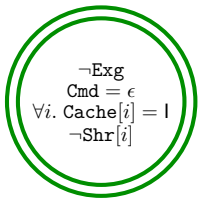
      ...

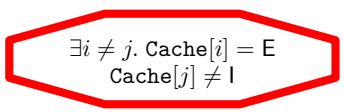
      return ψ

    ...

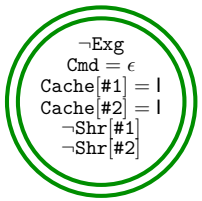
  return `pre`$_{\mathcal{T}}$(φ)

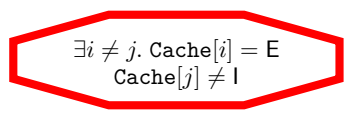
Example: BRAB on German-*ish*



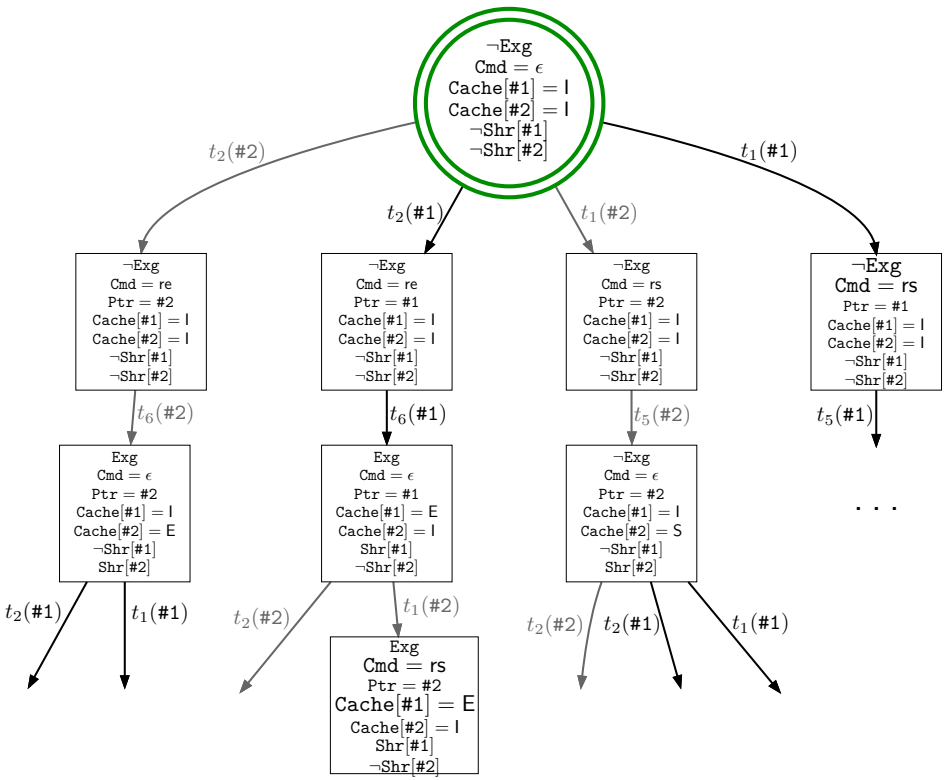


Example: BRAB on German-*ish*



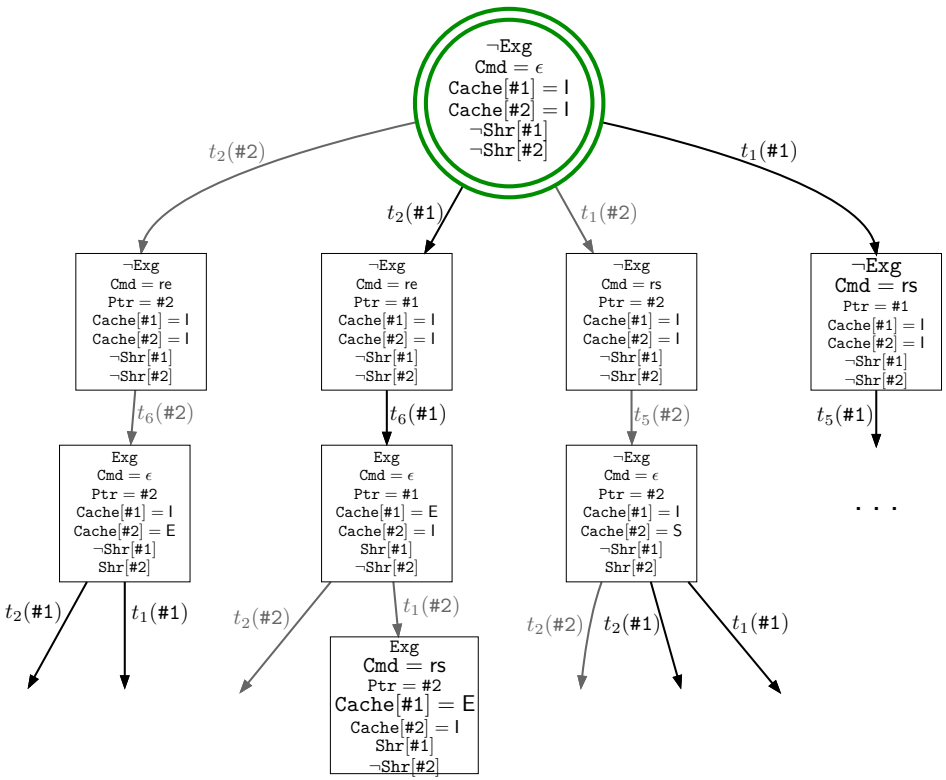


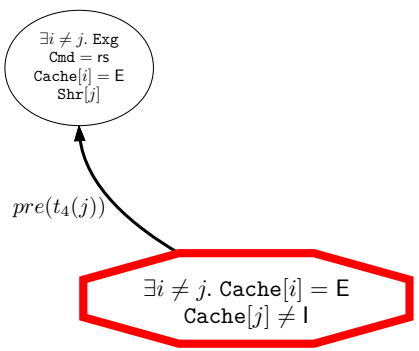
Example: BRAB on German-*ish*



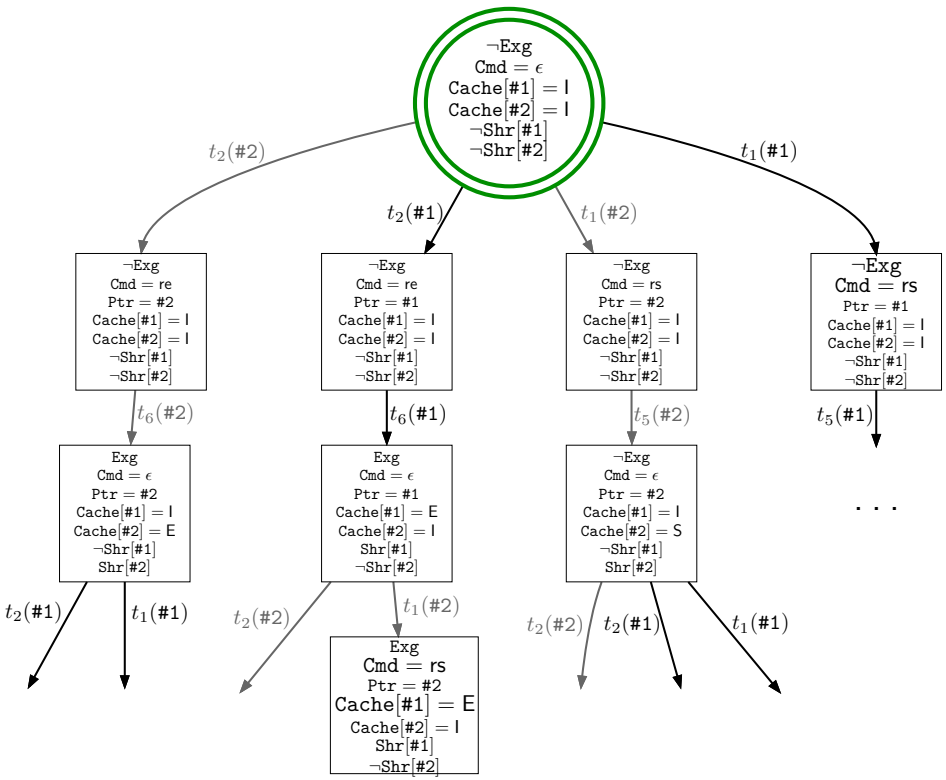
$\exists i \neq j.\ \mathsf{Cache}[i] = \mathsf{E}$
$\mathsf{Cache}[j] \neq \mathsf{I}$

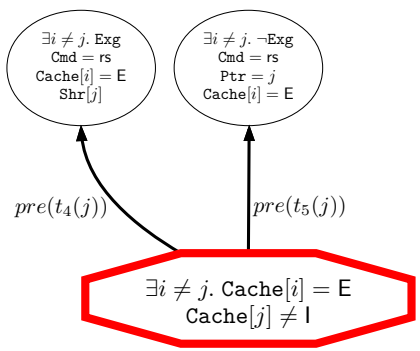
Example: BRAB on German-*ish*



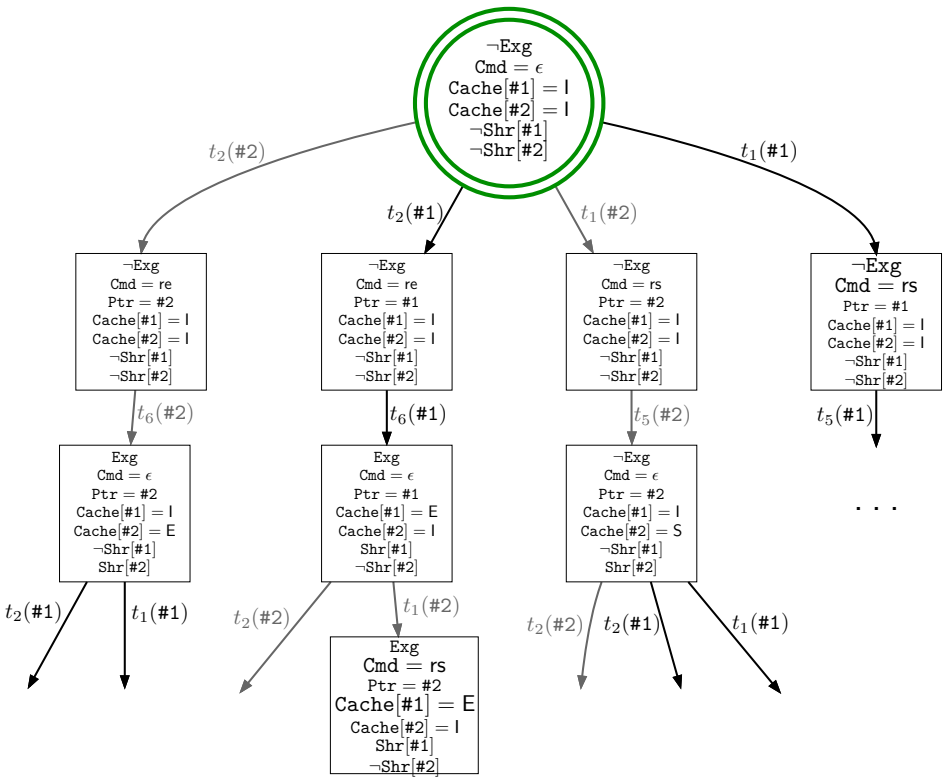


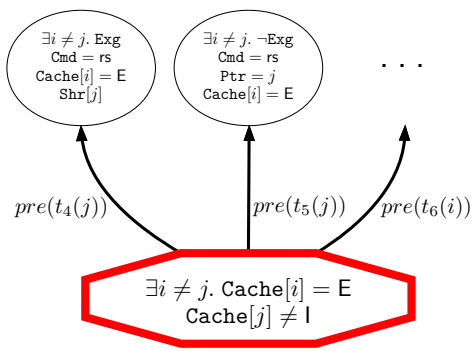
Example: BRAB on German-*ish*



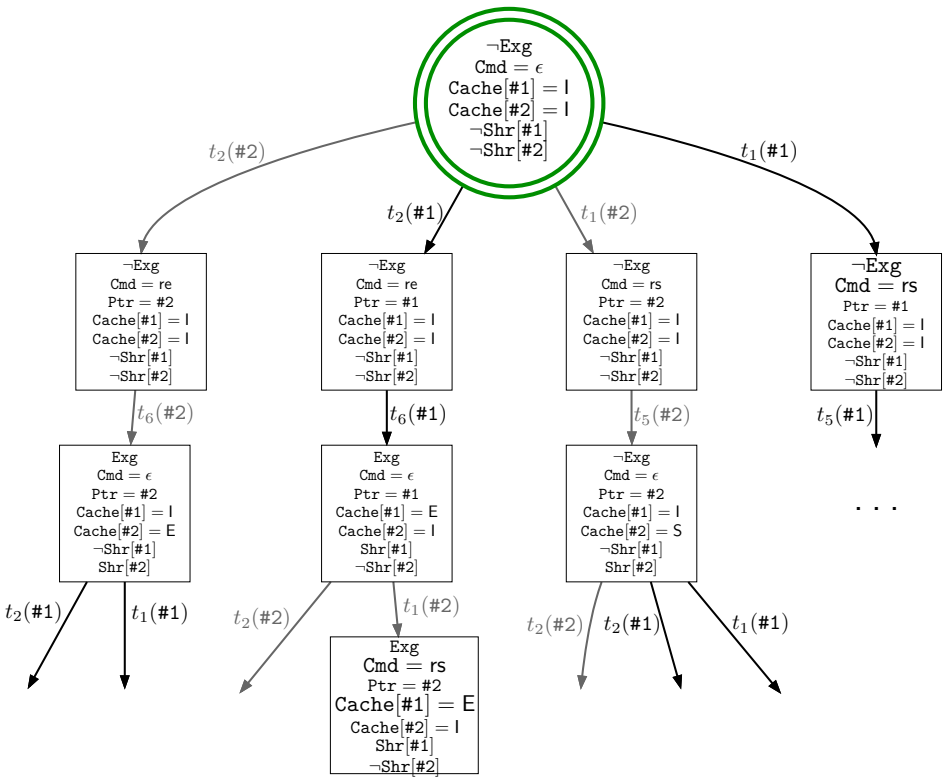


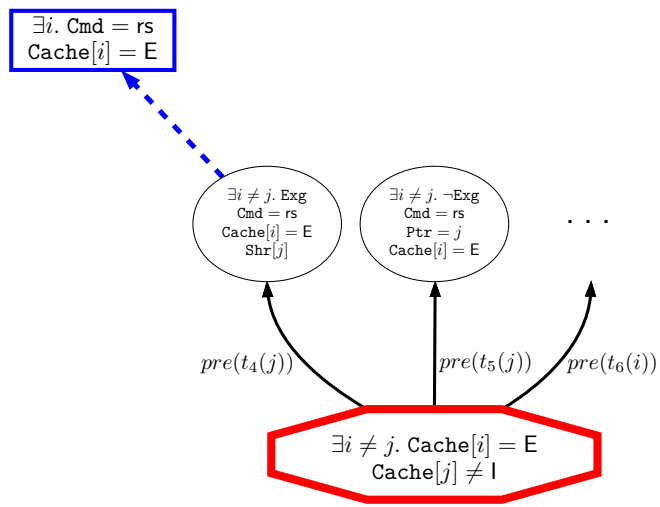
Example: BRAB on German-*ish*



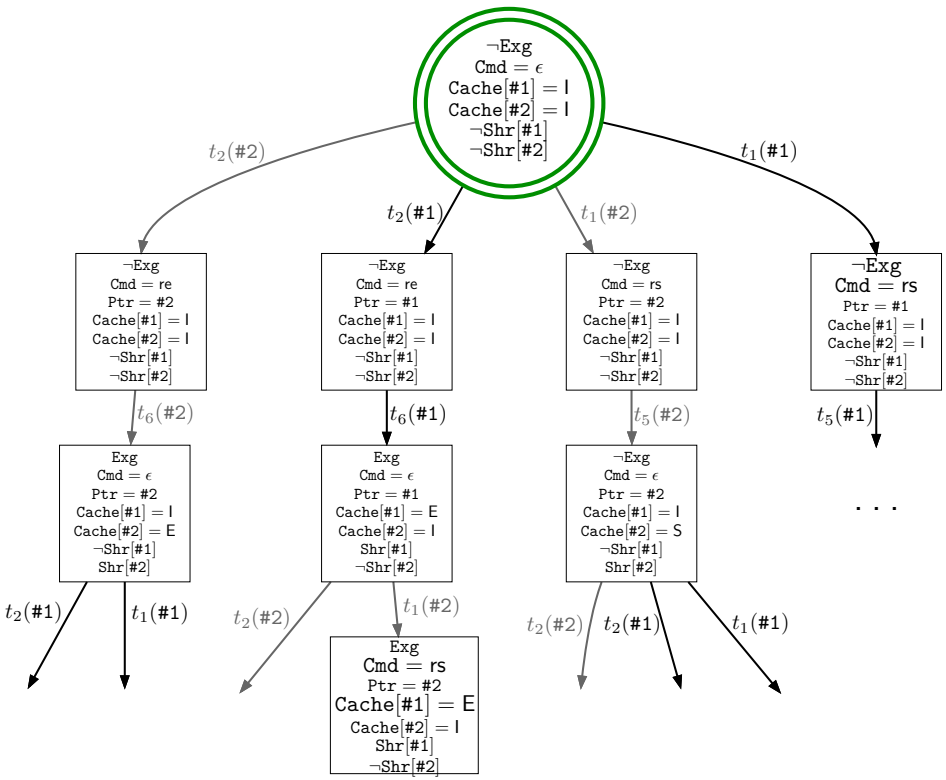


Example: BRAB on German-*ish*

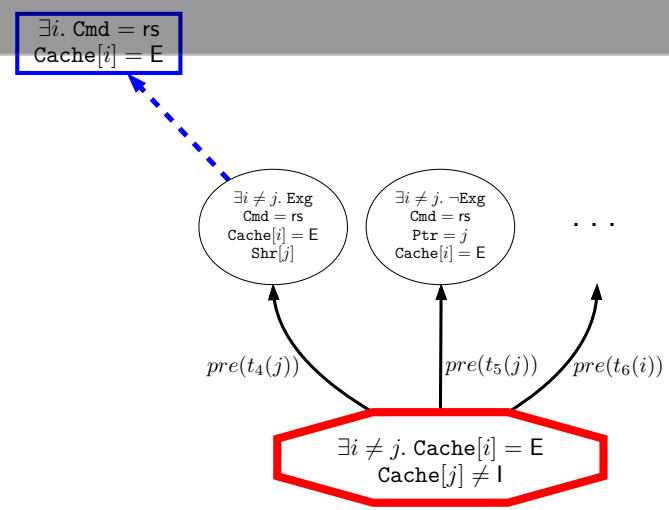




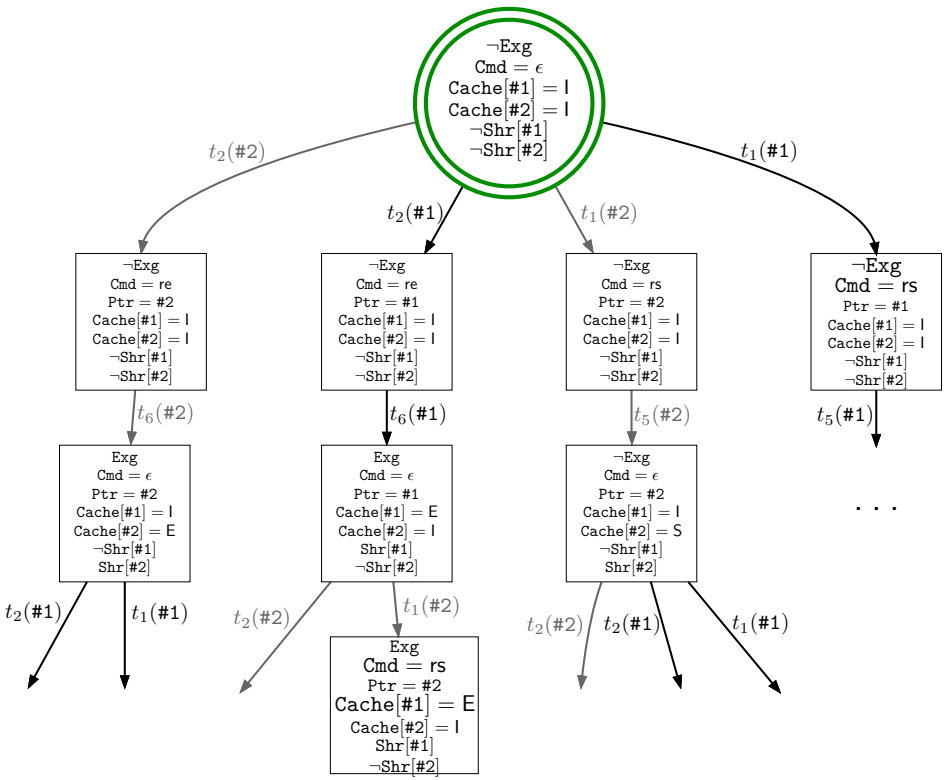
Example: BRAB on German-*ish*

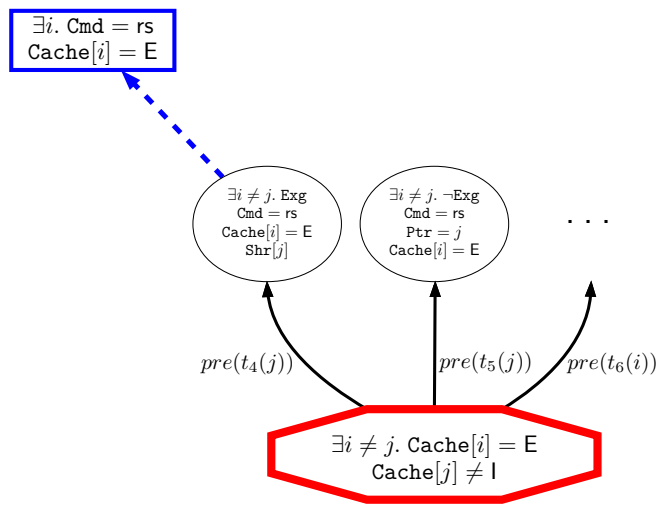


Extracting a cand



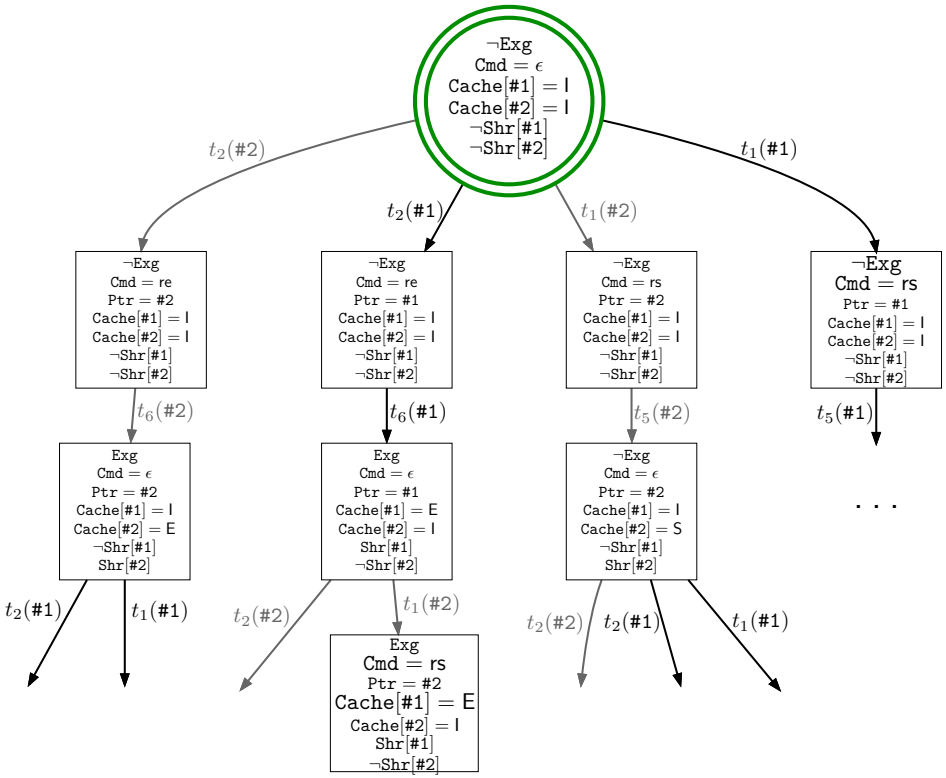
Example: BRAB on German-*ish*

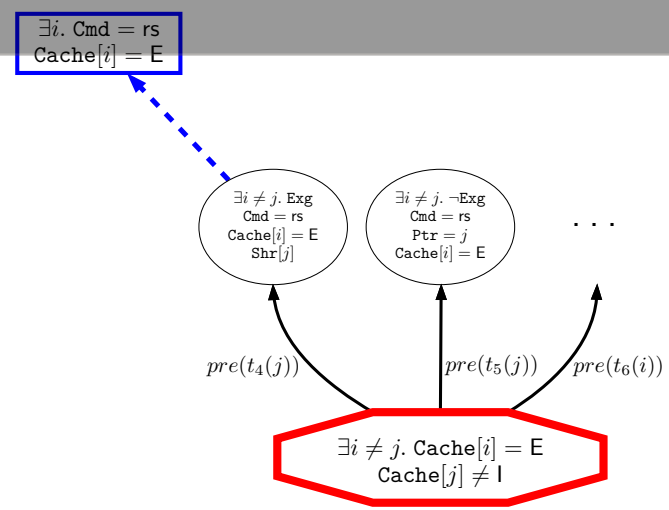




Example: BRAB on German-*ish*

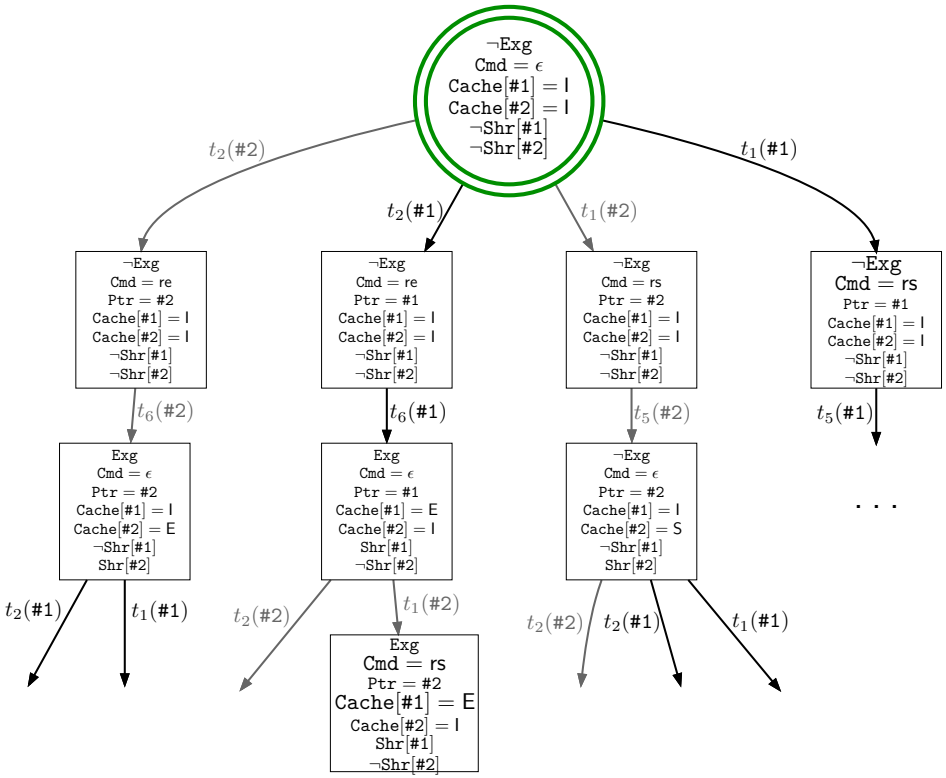
Checking

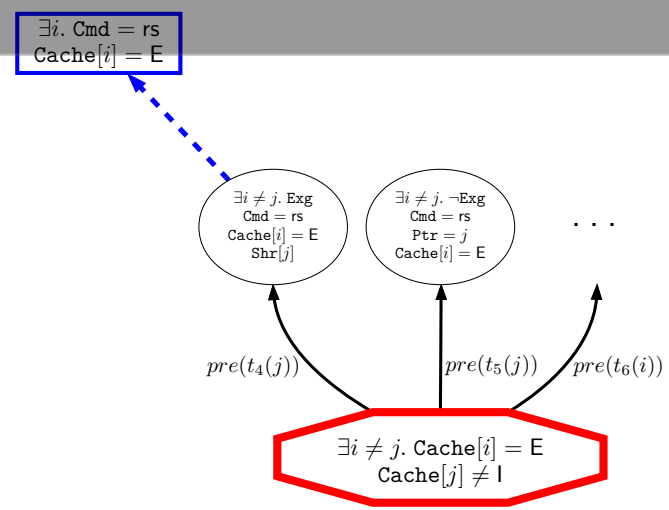




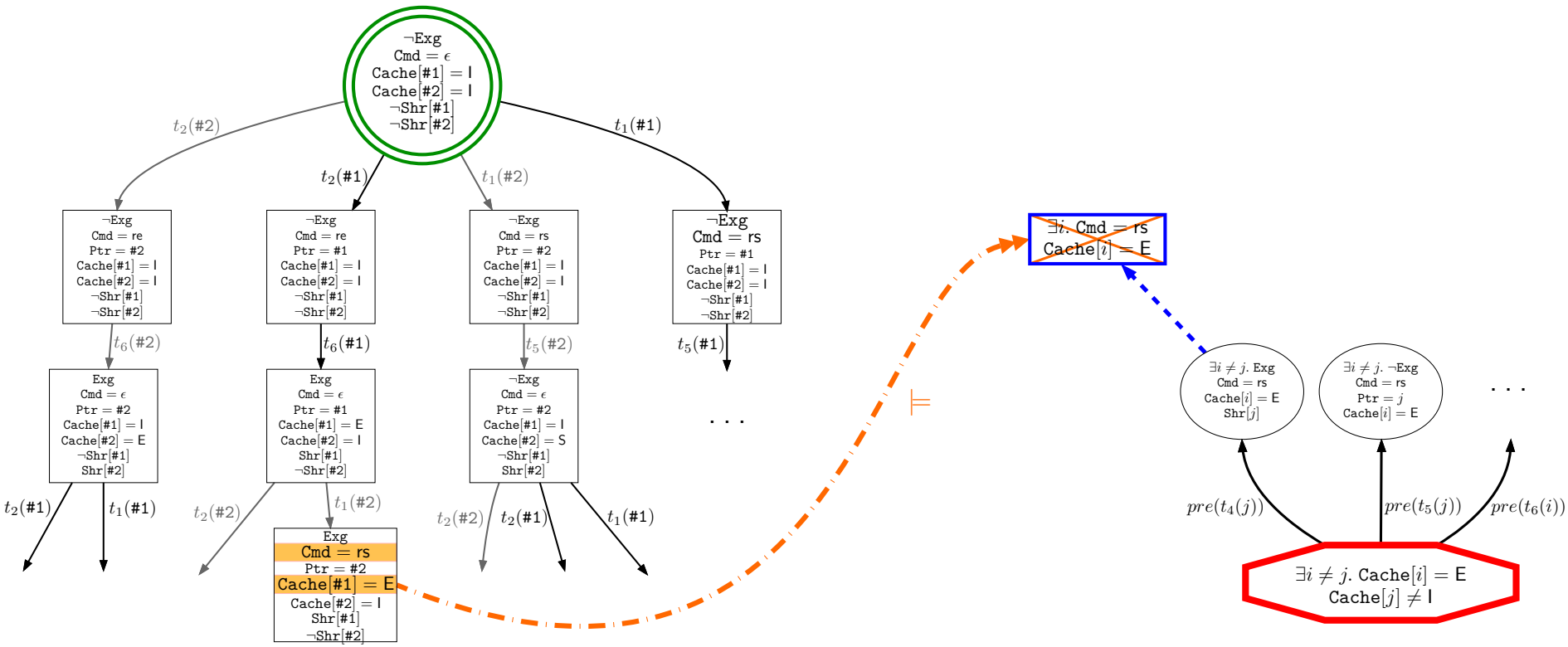
Example: BRAB on German-*ish*

Checking

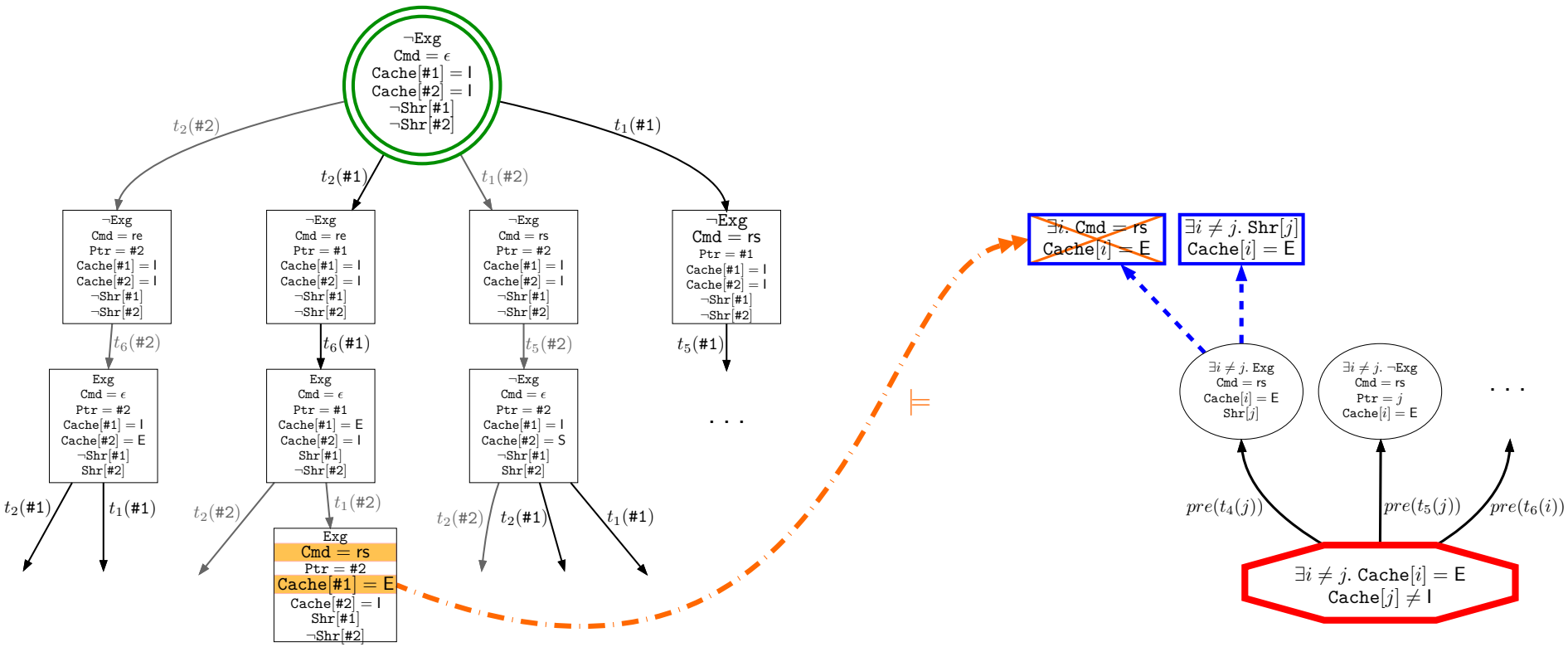




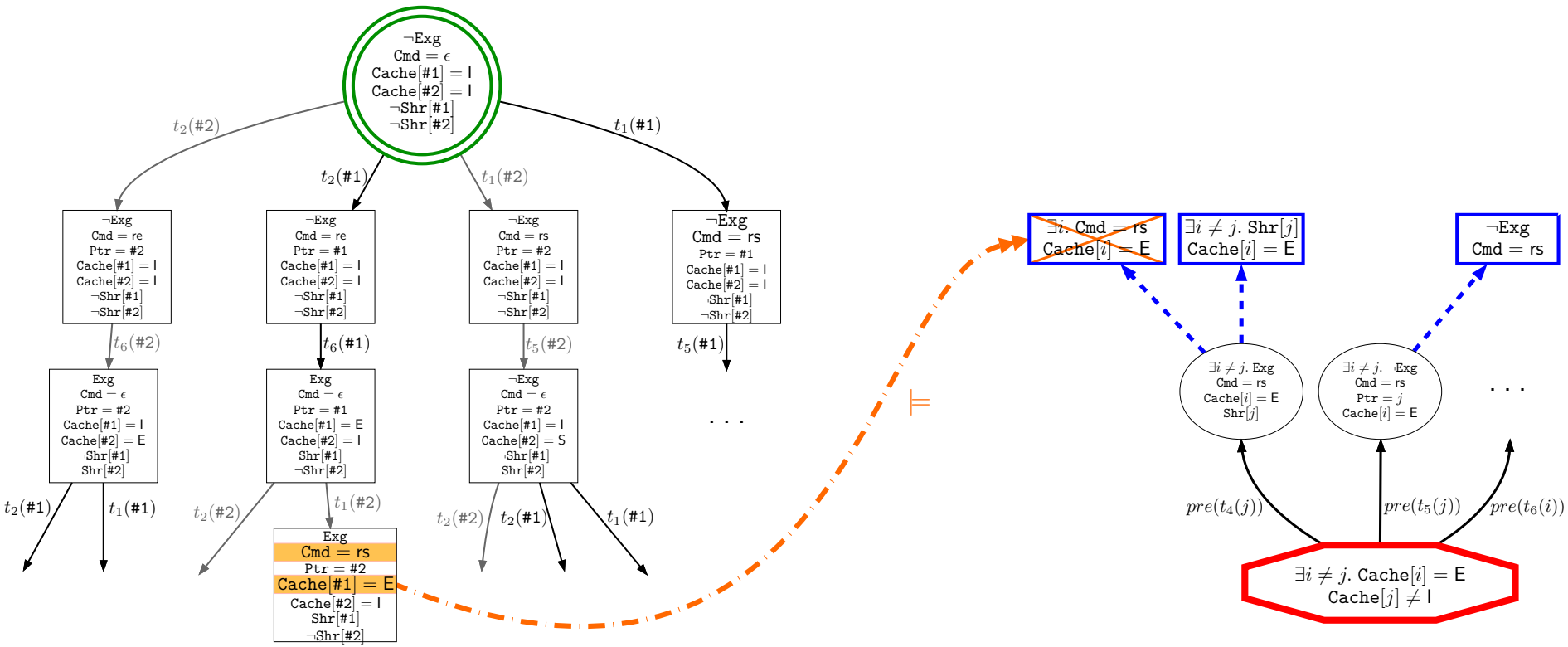
Example: BRAB on German-*ish*



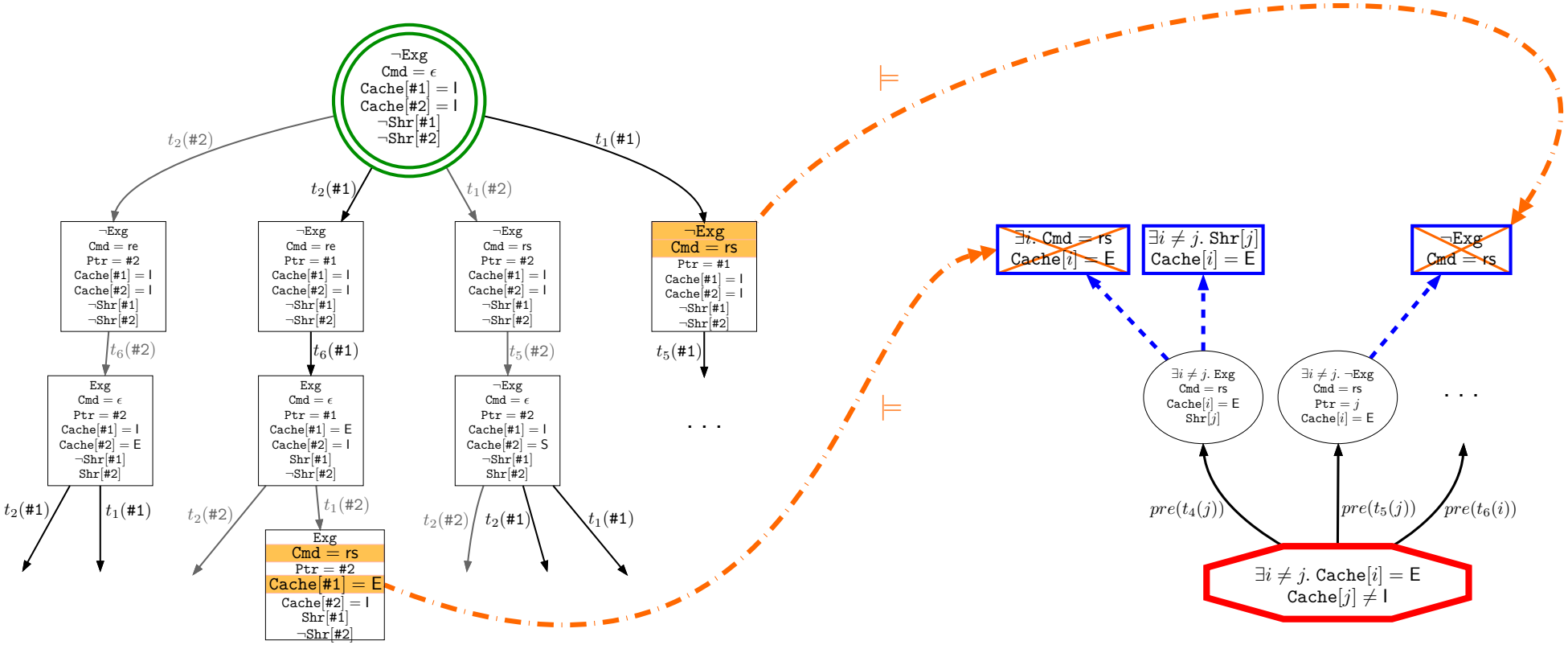
Example: BRAB on German-*ish*



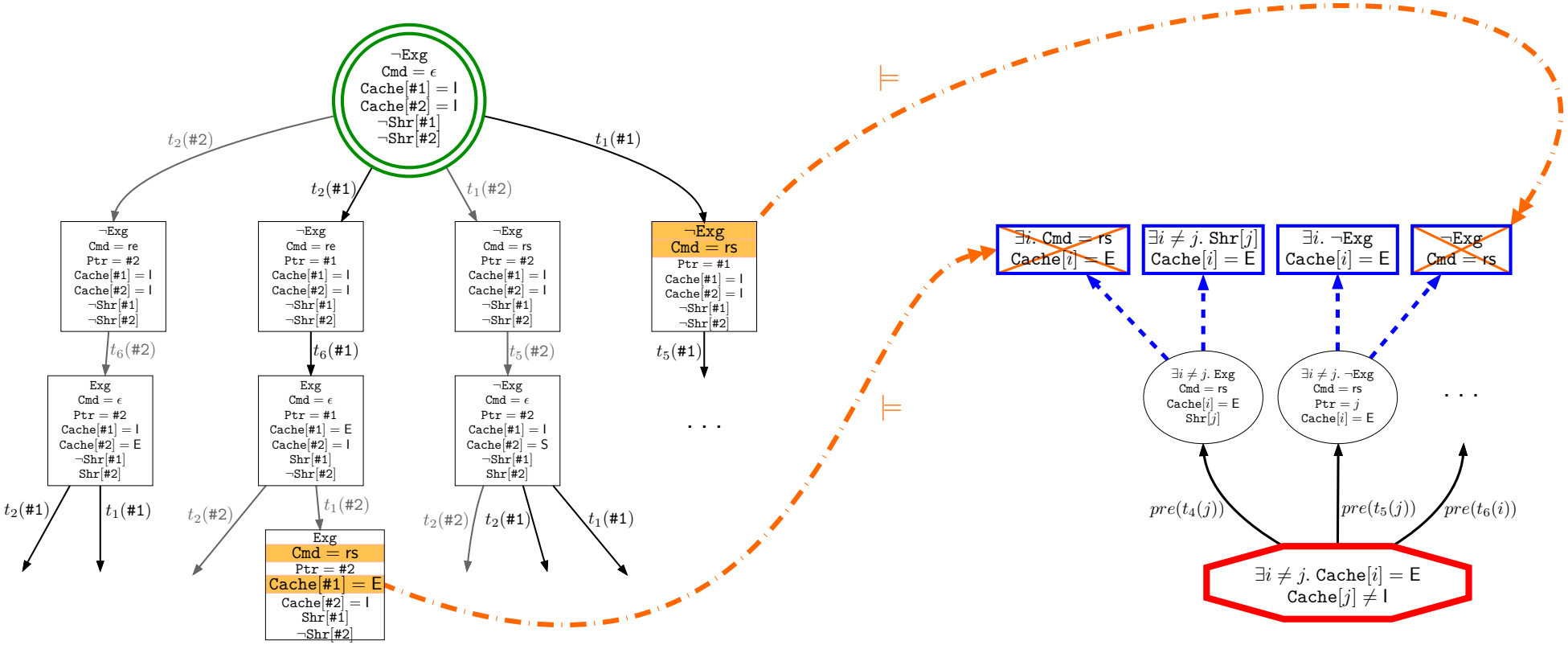
Example: BRAB on German-*ish*



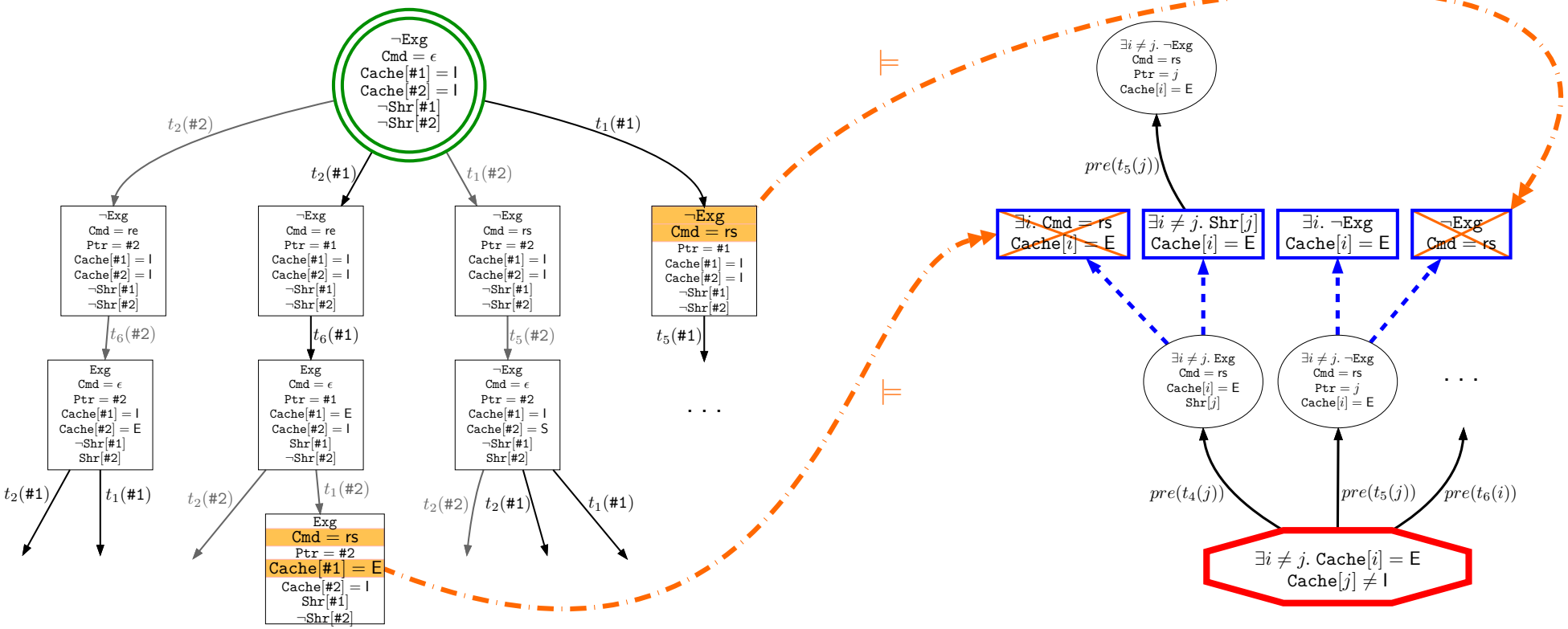
Example: BRAB on German-*ish*



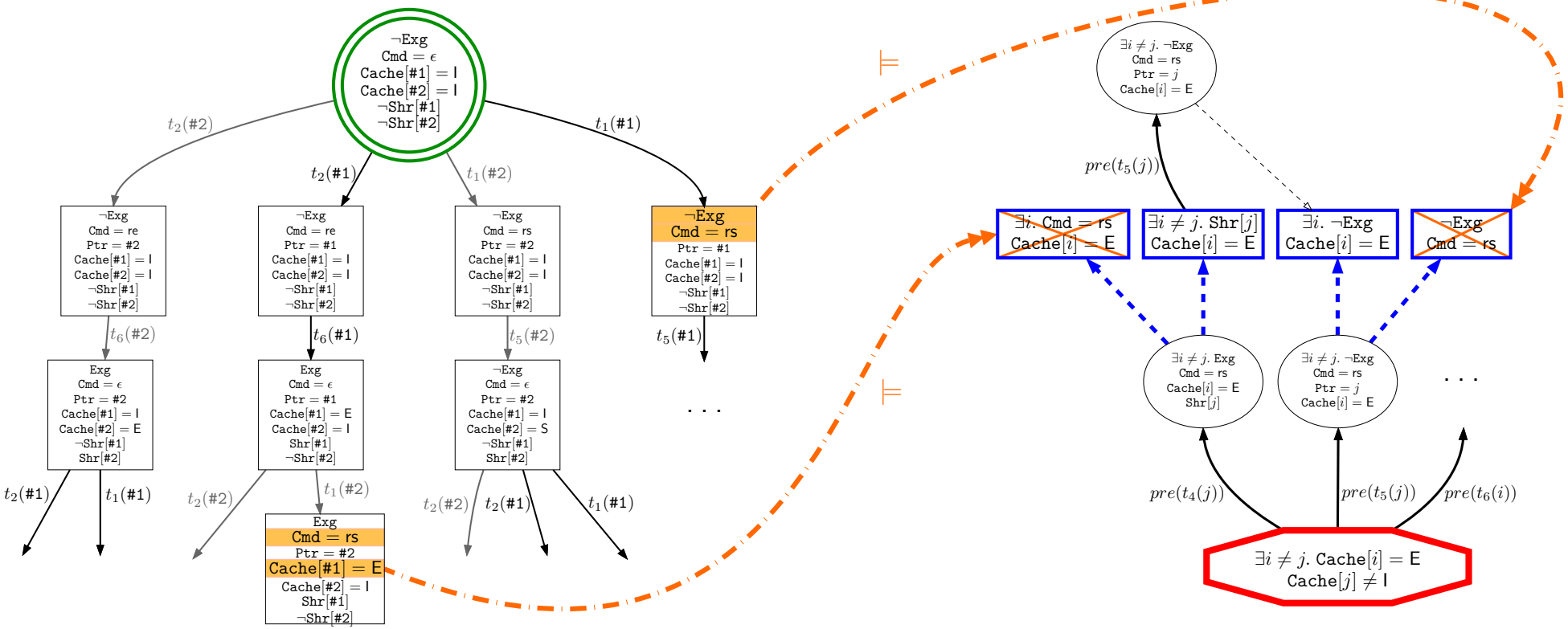
Example: BRAB on German-*ish*



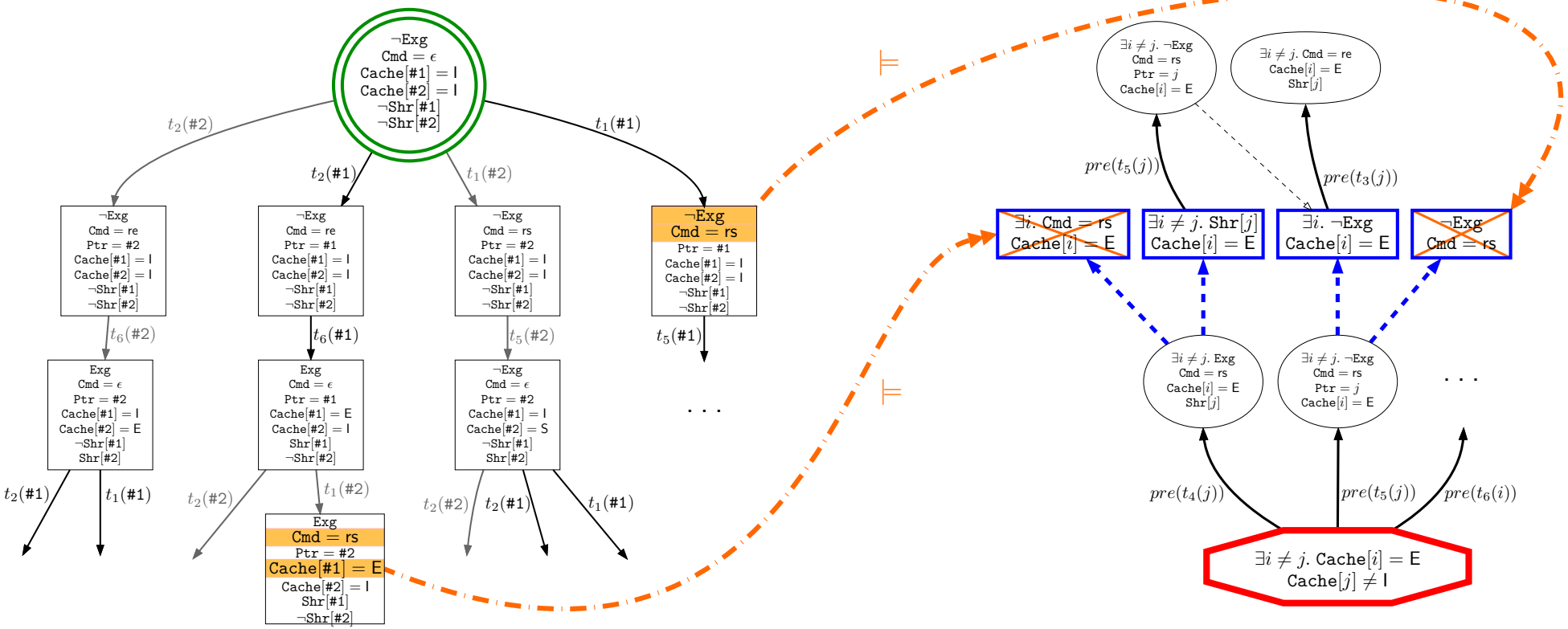
Example: BRAB on German-*ish*



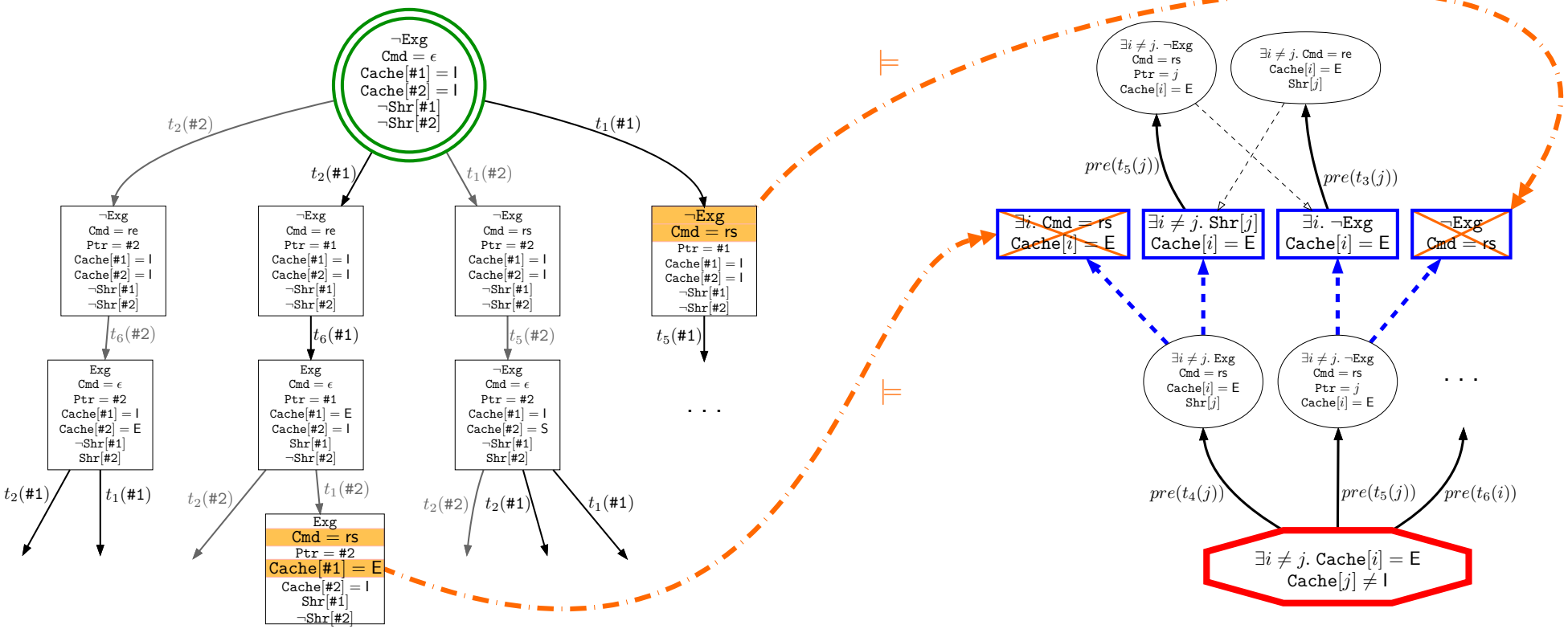
Example: BRAB on German-*ish*



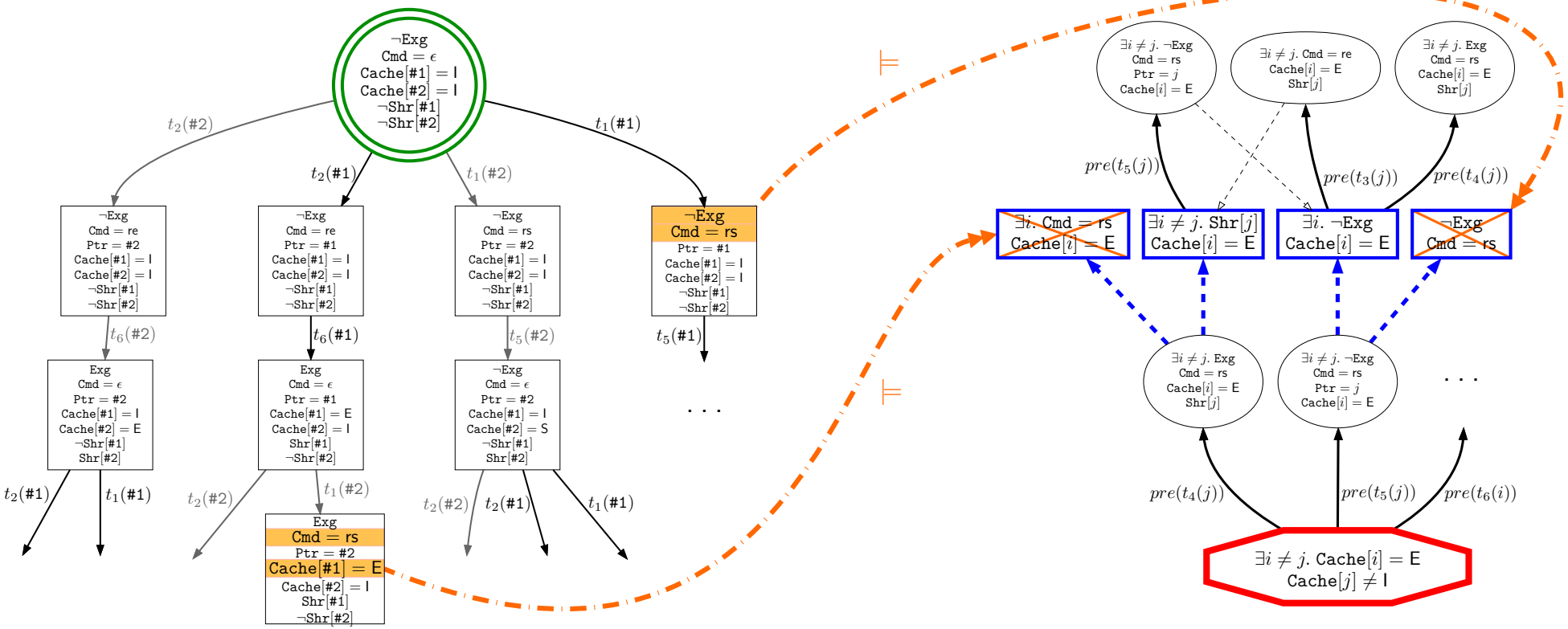
Example: BRAB on German-*ish*



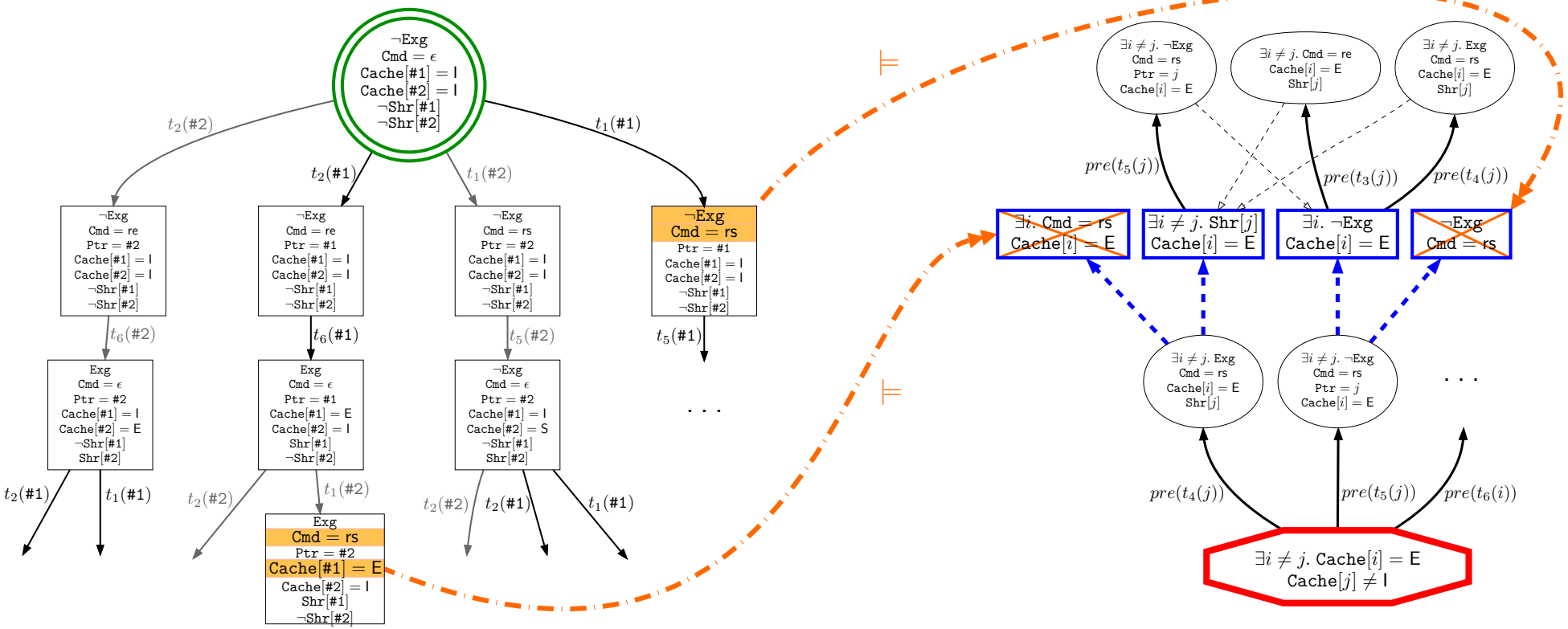
Example: BRAB on German-*ish*



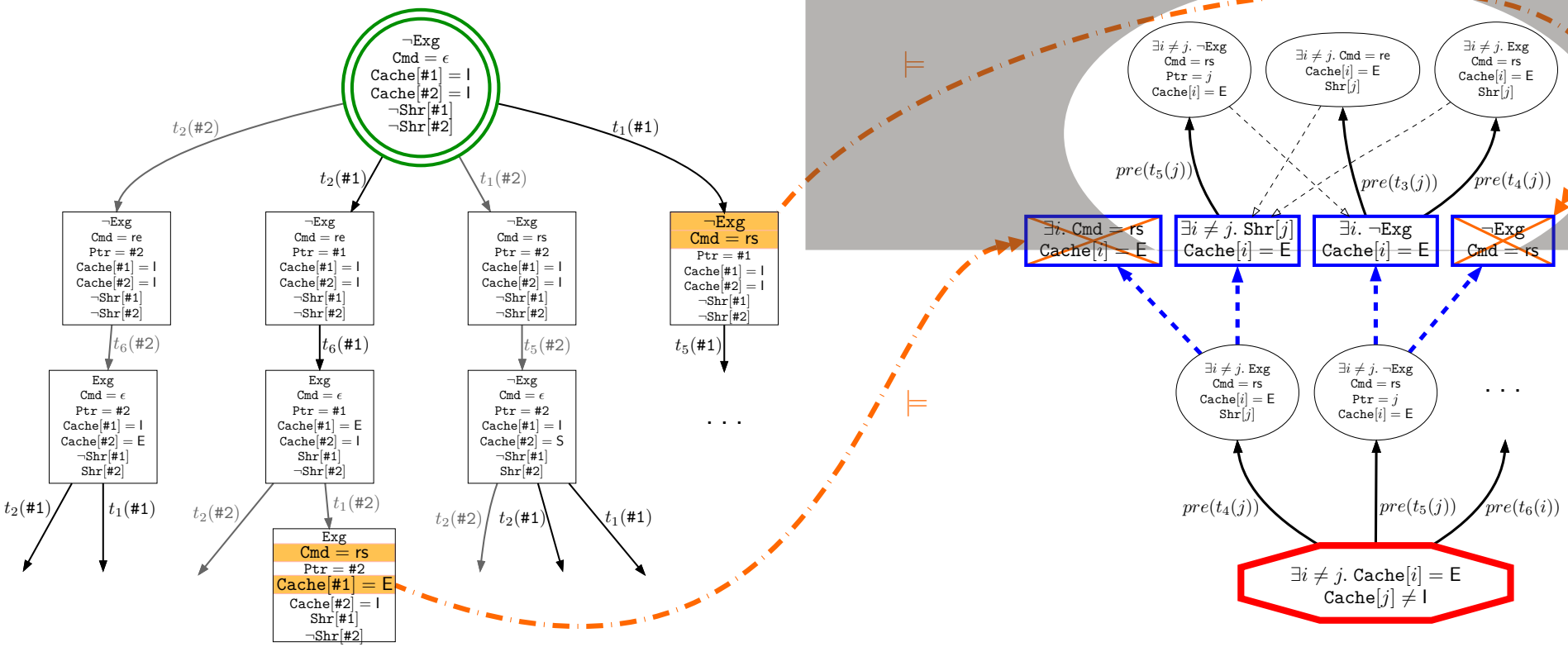
Example: BRAB on German-*ish*



Example: BRAB on German-*ish*



Example: BRAB on German-*ish*



Some benchmarks

	BRAB	Cubicle	CMurphi		
Szymanski_at	**0.14s**	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	**0.25s**	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	**0.29s**	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	**0.34s**	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	**2m17s**	2h01m	5.68s (4)	2m58s (5)	1h36m (6)
Szymanski_na	**0.19s**	T.O.	0.88s (4)	8m25s (6)	7h08m (8)
Flash_nodata	**0.36s**	O.M.	4.86s (3)	3m33s (4)	2h46m (5)
Flash	**5m40s**	O.M.	1m27s (3)	2h15m (4)	O.M. (5)

O.M. $>$ 20 GB

T.O. $>$ 20 h

Some papers

The Model Checking Modulo Theories paradigm has been originally proposed by Silvio Ghilardi et Silvio Ranise

- `http://users.mat.unimi.it/users/ghilardi/mcmt/`

In particular, I recommend

Backward Reachability of Array-based Systems by SMT Solving: Termination and Invariant Synthesis [LMCS, vol.6, n.4, 2010]

More information about Cubicle

In English :

- Certificates for Parameterized Model Checking [FM 2015]
- Invariants for Finite Instances and Beyond [FMCAD 2013]
- Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems [CAV 2012]

In french :

- Inférence d'Invariant pour le *model checking* de systèmes paramétrés [Alain Mebsout, PhD thesis, 2014]
- Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières [JFLA 2014]
- Vérification de systèmes paramétrés avec Cubicle [JFLA 2013]