동시 실행 프로그램 정적 검증기 설계

Design of Automatic Verifier for Concurrent Programs

정승철
한양대학교 프로그램 분석검증 연구실

쓰레드 등을 사용하여 동시에 실행되는 프로그램에서

공유된 메모리 영역을

동시에 접근하는 경쟁 상황이 발생하지 않는다는 것을

자동 증명할 수는 없을까?

대상 언어 - C 같은 언어

- 힙 메모리를 사용하는 C 같은 언어

```
x = E;
x = *E;
*E = F;
E = alloc(...);
dispose(E);
while (B) {...}
if (B) {...} else {...}
```

대상 언어 - 동시 실행 요소

- 메모리를 공유하면서 동시 실행

$$C_1 \;||\; C_2$$

- 락과 조건으로 보호받는 영역

locks

conditional critical region

```
with r when B { ... }
```

- 자원 `r` 을 소유하고 조건 `B` 를 만족하면 영역 진입
- 아니면 자원을 포기하고 다시 시도
- `{ ... }` 영역을 빠져나오면서 자원을 돌려줌

엇갈린 두 이진 세마포어

```
f = 1; b = 0; p = alloc();
```

f=1 && b=0 && p->-

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

엇갈린 두 이진 세마포어

```
f = 1; b = 0; p = alloc();
```

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;
```

f=0 && b=0 && p->-

```
  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

엇갈린 두 이진 세마포어

```
f = 1; b = 0; p = alloc();
```

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
```

f=0 && ***b=1*** && p->-

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

엇갈린 두 이진 세마포어

```
f = 1; b = 0; p = alloc();
```

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

f=0 && ***b=0*** && p->-

엇갈린 두 이진 세마포어

```
f = 1; b = 0; p = alloc();
```

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

f=1 && b=0 && p->-

엇갈린 두 01 세마포어

- 공유되는 `p` 셀이 **`with` 밖**에서 사용됨
- 세마포어를 가지고 실행 순서를 고정시켜 p 가 항상 한 쪽에서만 쓰이도록 만듦

```
f = 1; b = 0; p = alloc();
```

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }
  *p = x;
  with b when true {
    b = b + 1;
  }
}
```

||

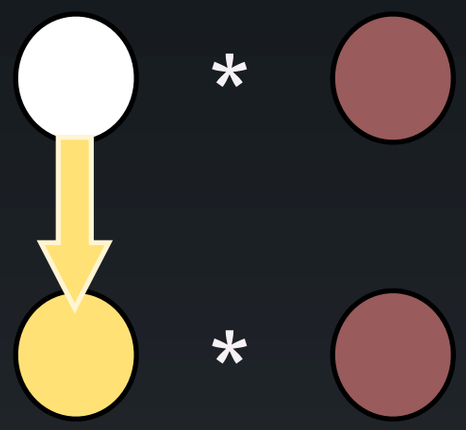
```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }
  y = *p;
  with f when true {
    f = f + 1;
  }
}
```

동시 분리 논리

Concurrent Separation Logic

동시분리논리

- 분리논리
  - 분리곱 * 을 사용해 메모리의 분리되었음을 표현
  - 메모리가 분리된 경우 한 쪽을 수정하여도 다른 쪽은 아무런 영향이 없음



- 동시분리논리 - 동시 실행 프로그램 증명에 적용

메모리 나누어 사용

- 분리곱 * 을 사용하면 메모리가 완전히 분리되어 있음을 표현 가능
- 나누어진 메모리를 각 프로세스에게 분배하면 다른 프로세스를 고려하지 않고 증명가능

C 는 P’ 을 건드리지 않음

$$\frac{\{P\}C\{Q\} \qquad \{P'\}C'\{Q'\}}{\{P * P'\}C \parallel C'\{Q * Q'\}}$$

P 와 P’ 은 완전히 분리된 영역

자원불변식 resource invariants

- `with` 블록을 통해서만 접근 가능한 메모리를 기술하는 불변식
- 프로세스가 마음대로 사용할 수 없는 일종의 관리되는 메모리를 기술

관리되는 메모리는
메모리는 남겨 놓아야 함

관리되는 메모리를 C 가 접근 가능

$$\frac{\{(P * RI_r) \wedge B\}C\{Q * RI_r\}}{\{P\}\texttt{with } r \texttt{ when } B \texttt{ \{ } C \texttt{ \}}\{Q\}}$$

예. 조심스럽게 with 안에서만

RI_r : l->-

l 힙 셀은 관리됨

emp 사용할 수 있는 메모리는 없음

```
with r when true {

  *l = x;

}
```

||

```
with r when true {

  y = *l

}
```

emp

예. 조심스럽게 with 안에서만

RI_r : l->-

emp

emp

l을 얻음

l->-

```
with r when true {
  *l = x;
}
```

l->-

l을 돌려줌

emp

||

emp

l을 얻음

l->-

```
with r when true {
  y = *l;
}
```

l->-

l을 돌려줌

emp

emp

예. 무모한 도전

RI_r : l->-

emp

emp

```
with r when true {
```

l->-

```
   *l = x;
```

l->-

```
}
```

emp

||

emp

```
y = *l;
```

emp

소유권 분쟁

- 분리논리 항은 **해당 지점의 프로세스만이 소유한 메모리**를 기술
- 동시분리논리는 특정 메모리 영역을 항상 다음 중 한 곳만이 가질 수 있도록 함
  - 프로세스 중 하나 (사용자)
  - 혹은 자원불변식 (관리자)
- with 문을 지나면서 메모리 소유권이 바뀔 수 있음
- 소유권 분쟁을 적절히 조정하는 자원불변식을 찾는 것이 증명의 관건

메모리 임대

- 자원불변식을 조건식으로 기술하면?

```
(f=1 && p->-) || (f=0 && emp)
```

  - `f` 가 0 인 경우에는 힙 셀을 회수하지 않음
  - `with` 안에서 `f` 를 0으로 바꾸면 힙 셀을 밖으로 전달 가능
- 관리자가 사용자에게 메모리를 빌려줌
- `with` 밖에서 공유 메모리 접근 가능

엇갈린 두 이진 세마포어

RI_f : (f=1 && p->-) || (f=0 && emp)
RI_b : (b=1 && p->-) || (b=0 && emp)

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

소유권 획득

RI_f : (f=1 && p->-) || (f=0 && emp)
RI_b : (b=1 && p->-) || (b=0 && emp)

프로세스가 관리되던 p를 얻어냄

```
emp
with f when f > 0 {
                ((f=1 && p->-) || (f=0 && emp)) && f>0
                f=1 && p->-
    f = f - 1;
                f=0 && p->-
                f=0 && (p->- * emp)
                ((f=1 && p->-) || (f=0 && emp)) * (f=0 && p->-)
}
p->-
```

엇갈린 두 이진 세마포어

RI_f : (f=1 && p->-) || (f=0 && emp)
RI_b : (b=1 && p->-) || (b=0 && emp)

emp

p->-

```
while (true) {

  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }

}
```

||

emp

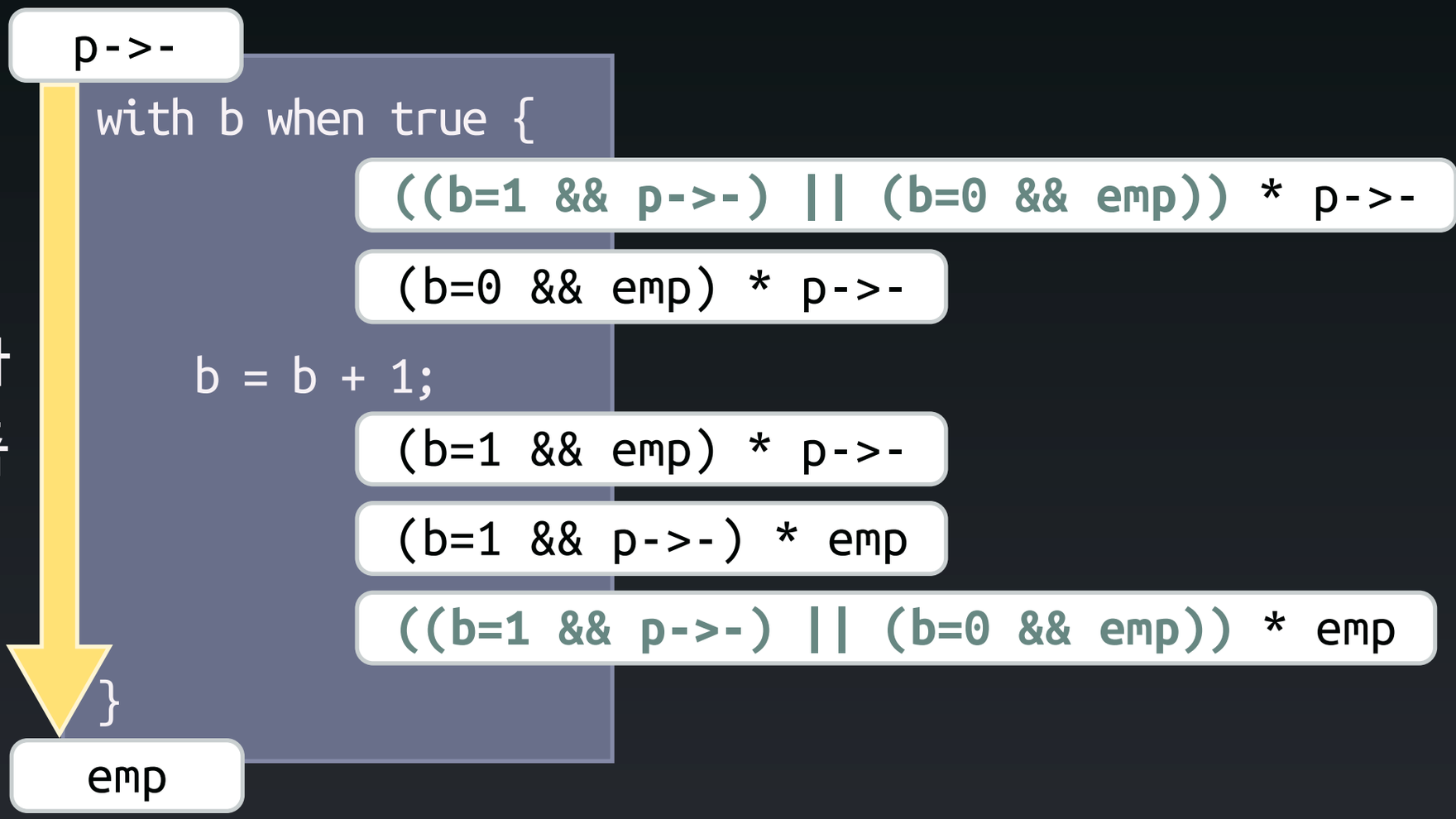
p->-

```
while (true) {

  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }

}
```

소유권 반환

RI_f : (f=1 && p->-) || (f=0 && emp)
RI_b : (b=1 && p->-) || (b=0 && emp)

프로세스가
p를 돌려줌



엇갈린 두 이진 세마포어

RI_f : (f=1 && p->-) || (f=0 && emp)
RI_b : (b=1 && p->-) || (b=0 && emp)

emp

```
while (true) {

  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }

}
```

p->-

p->-

emp

||

```
while (true) {

  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }

}
```

emp

p->-

p->-

emp

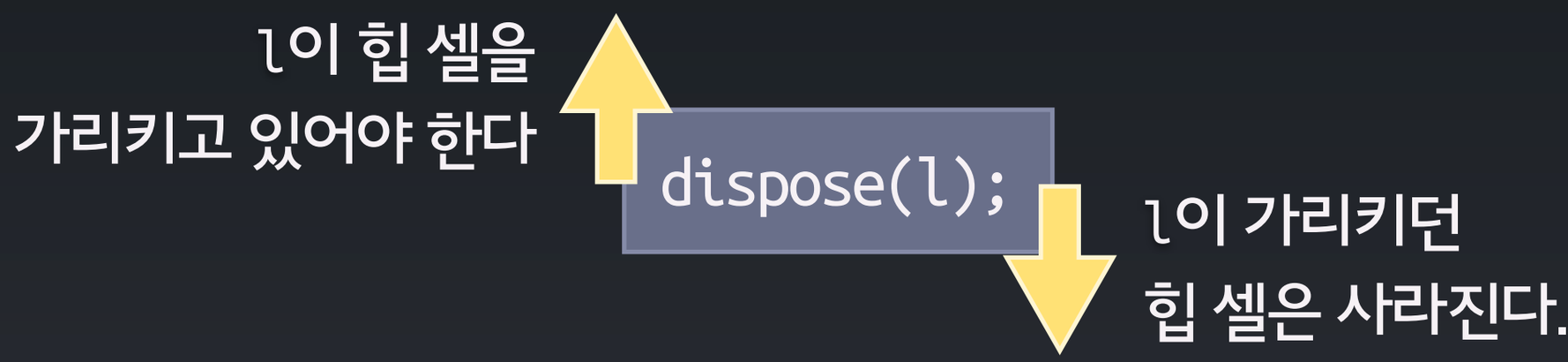
자동 검증기 설계

자동 검증기 설계

- 목표 - 동시분리논리를 기반으로 프로그램을 자동으로 검증하는 시스템을 설계
- 접근 방향
  - 자원불변식을 주면 제대로 동작하는지 검사 (X)
  - 코드가 주어지면 자원불변식을 자동으로 찾고 제대로 동작하는지 검증 (O)
- 코드의 스펙을 끄집어내어야 함

양방향 분석 bi-directional analysis

- 분석 목표
  - 특정 코드가 안전하게 동작할 수 있는 전조건과
  - 그러한 전조건을 만족할 때 코드가 수행된 후 얻어지는 후조건을 찾아냄
- 새로운 전조건은 실행 역순으로 전파됨



메모리 모양 맞추기

- 양방향 분석의 입력
  - 아래쪽에서 새로 만들어져 올라온 전조건
  - 위쪽에서 새로 만들어져 내려온 후조건
  - 분석하고자 하는 구문
- 앞과 뒤의 메모리 모양을 맞추는 알고리즘 필요
  - Bi-Abductor

$$P * [R] \vdash Q * [S]$$

양방향 분석 규칙 - skip

$$\frac{P * [R] \vdash Q * [S]}{\{P * [R]\}\texttt{skip}\{Q * [S]\}}$$

위쪽에서 빨간색과 하얀색
힙 셀이 만들어졌어

아래쪽에서 파란색과 빨간색
힙 셀이 필요해

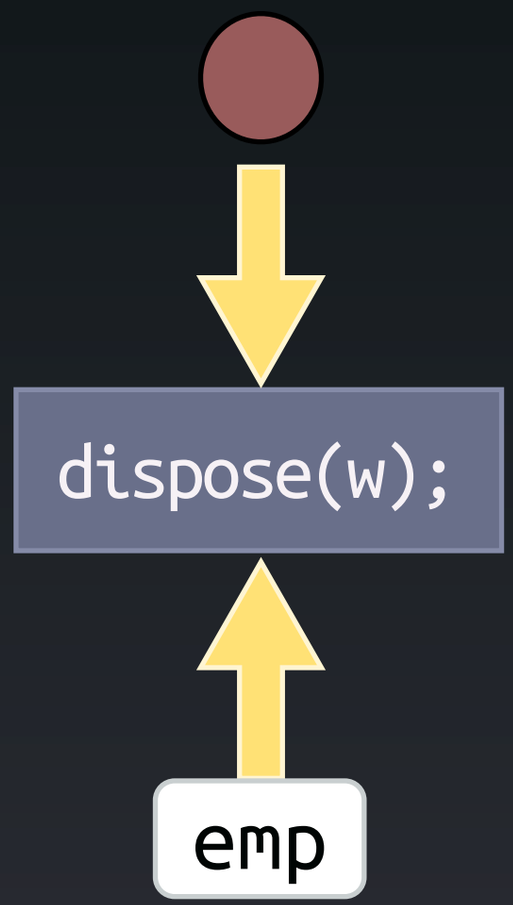
양방향 분석 규칙 - skip

$$\frac{P * [R] \vdash Q * [S]}{\{P * [R]\}\texttt{skip}\{Q * [S]\}}$$

양방향 분석 규칙 - dispose

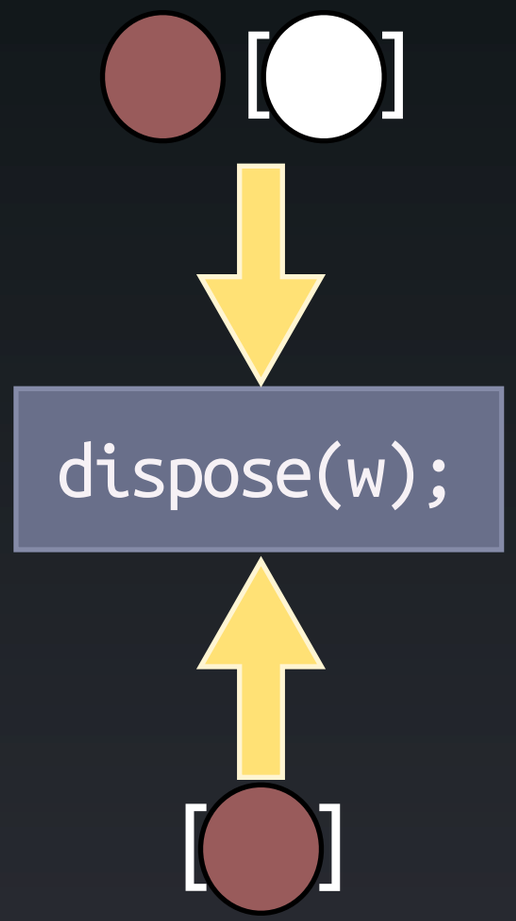
$$\frac{P * [R] \vdash Q * E \mapsto a * [S]}{\{P * [R]\}\texttt{dispose}(E)\{Q * [S]\}}\ \text{DISPOSE}$$

dispose(w)가 수행되려면
힙 셀 w 가 있어야 함



양방향 분석 규칙 - dispose

$$\frac{P * [R] \vdash Q * E \mapsto a * [S]}{\{P * [R]\}\texttt{dispose}(E)\{Q * [S]\}}\ \text{DISPOSE}$$



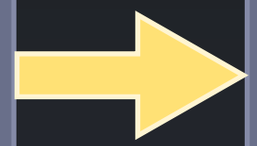
동시분리논리 적용

- 양방향 분석을 동시분리논리에 적용
- 전조건과 후조건 뿐만 아니라 자원불변식 역시 점진적으로 찾아냄

$$\frac{\ldots}{RI_r * [T] \vdash \{P * [R]\} C \{Q * [S]\}}$$

- 편의를 위해 `with` 를 분리

```
with r when B {
    ...
}
```

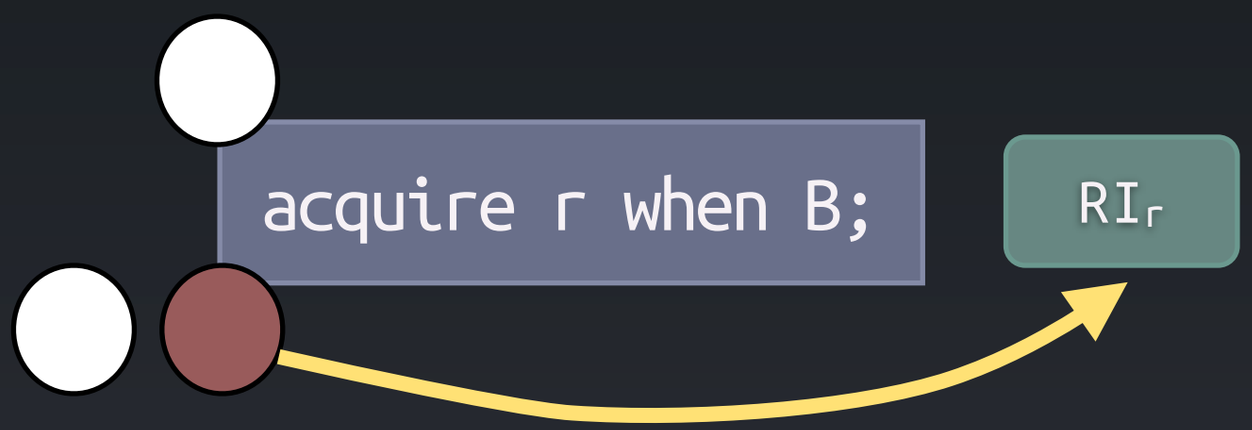


```
acquire r when B;
  ...
release r;
```

acquire 는 필요한 걸 내준다

- acquire 는 아래서 필요한 메모리를 내주어야 함
  - 자원불변식은 필요한 메모리를 지님
  - acquire 이전에는 필요한 메모리가 없어야 함

$$\frac{P * RI_r \wedge \Pi \wedge B * [R] \vdash Q \wedge \Pi * [S]}{RI_r * [B \Rightarrow R] \vdash \{P \wedge [\Pi]\}\texttt{acquire}\ r\ \texttt{when}\ B\{Q \wedge \Pi * [S]\}}$$



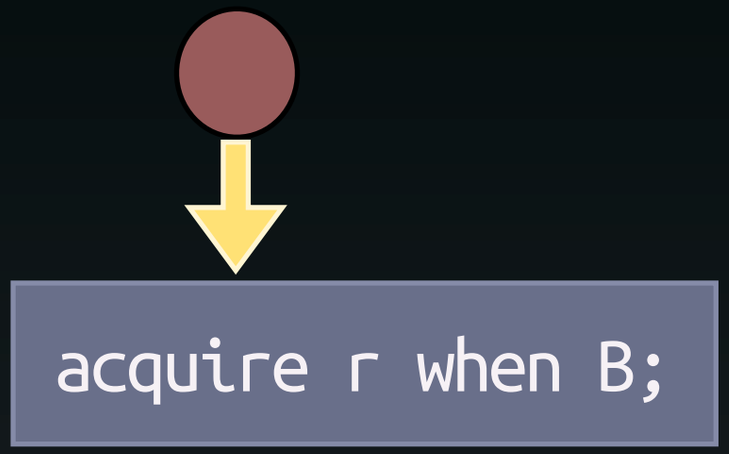
`release` 는 다 쓴 걸 걸 가져간다

- `release` 는 이후로 소유하지 않는 메모리를 가져감
  - 자원불변식은 필요한 메모리를 지님
  - `release` 다음에는 필요한 메모리가 없어야 함

$$\frac{P \land \Pi * [R] \vdash Q \land \Pi * RI_r * [S]}{RI_r * [\Pi \Rightarrow S] \vdash \{P \land \Pi * [R]\}\texttt{release}\ r\{Q \land [\Pi]\}}$$

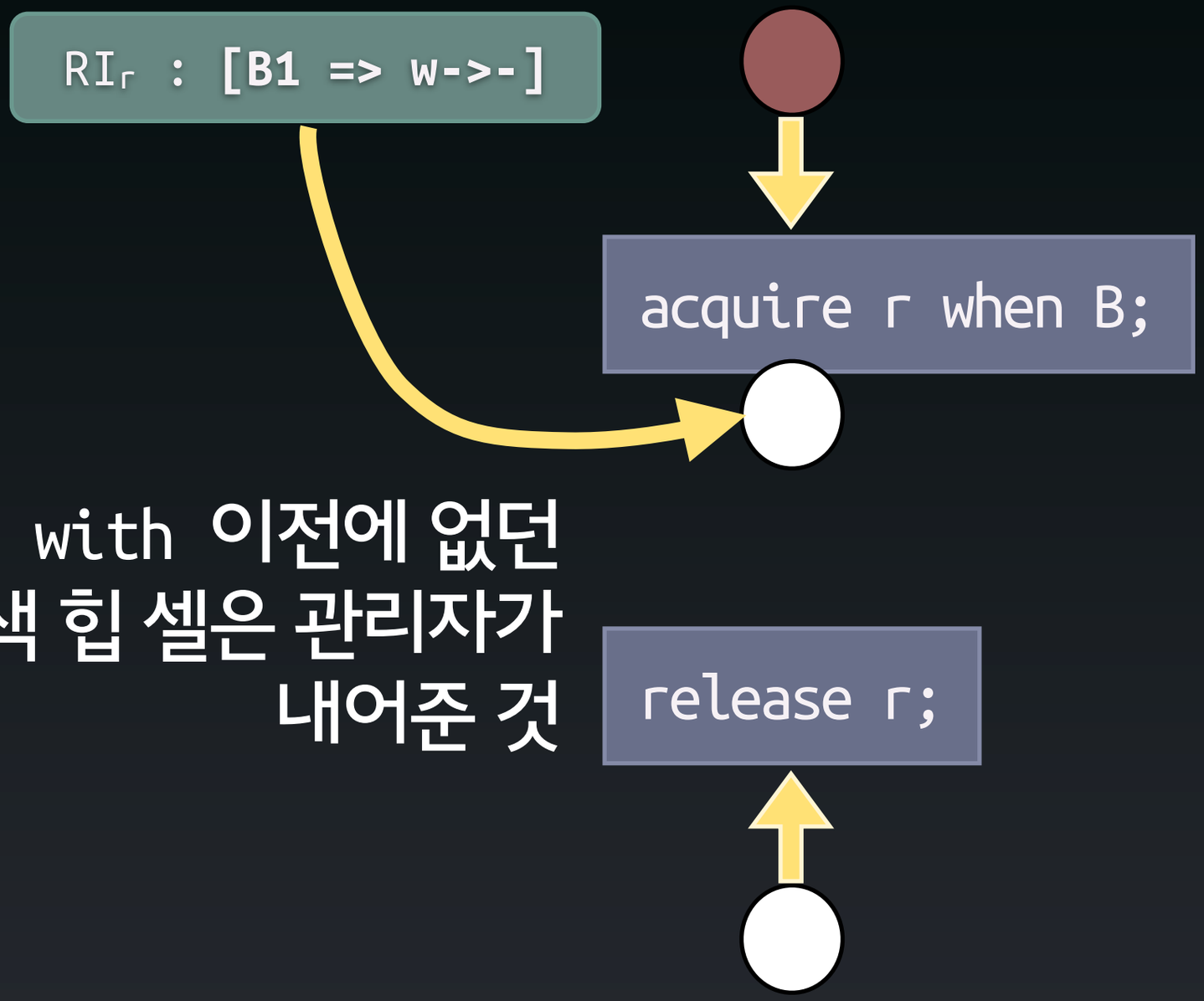
acquire…release

RI_r



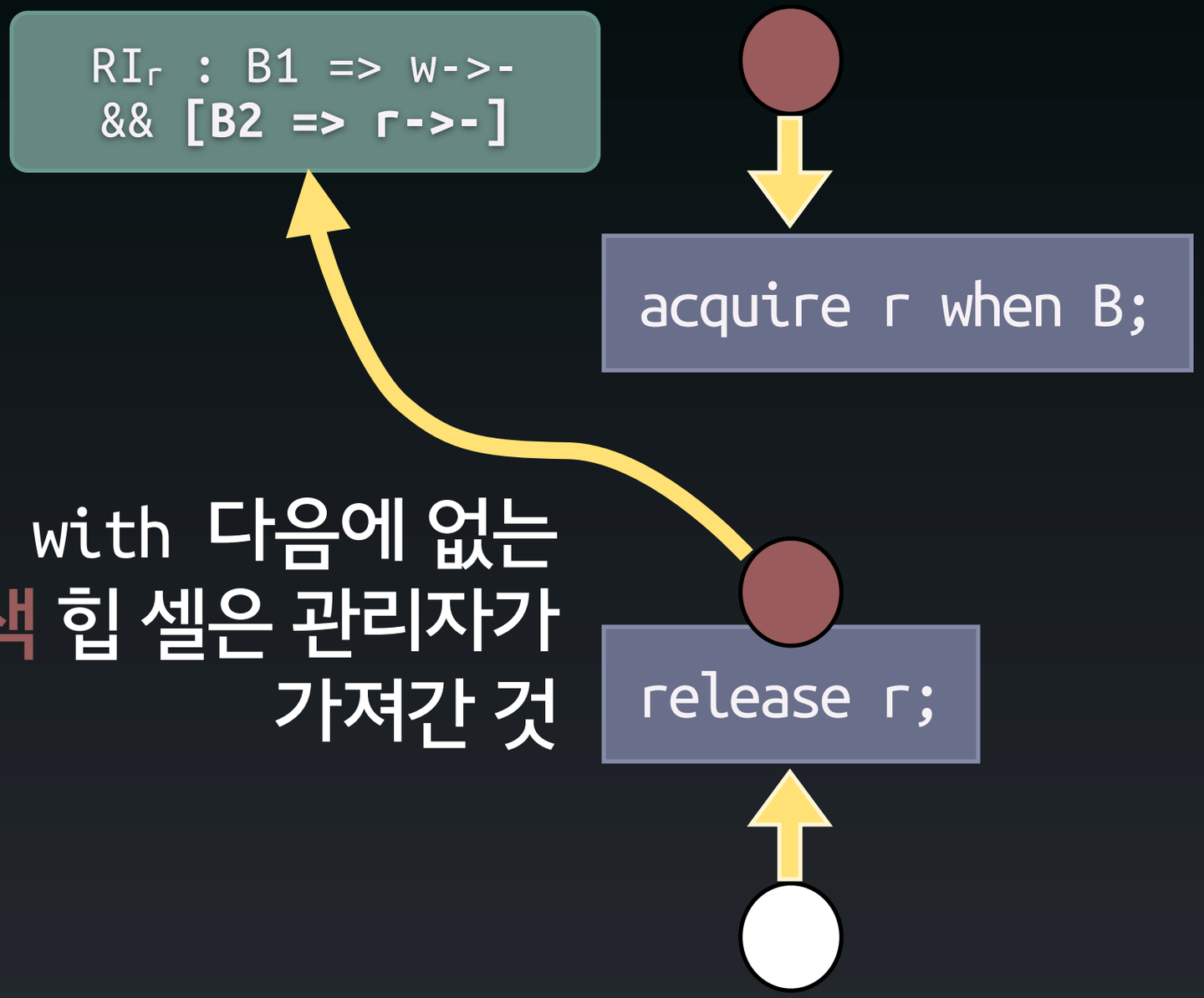
with 이전에는
빨간색 힙 셀이,
with 다음에는
하얀색 힙 셀이 있음

acquire…release



with 이전에 없던
하얀색 힙 셀은 관리자가
내어준 것

acquire…release



with 다음에 없는
빨간색 힙 셀은 관리자가
가져간 것

엇갈린 두 이진 세마포어

```
while (true) {
  with f when f > 0 {
    f = f - 1;
  }

  *p = x;

  with b when true {
    b = b + 1;
  }
}
```

||

```
while (true) {
  with b when b > 0 {
    b = b - 1;
  }

  y = *p;

  with f when true {
    f = f + 1;
  }
}
```

프로세스가 힙 셀을 사용

```
acquire f when f > 0;

  f = f - 1;


release f;

*p = x;
```

[p->?]

[p->?]

p에 힙 셀이
있어야 함

관리자가 힙 셀을 내어줌

RI_f : [f=1 => p->?]

여기서 관리자가
힙 셀을 내어줌

```
acquire f when f > 0;
```

f=1 && [p->?]

```
  f = f - 1;
```

f=0 && [p->?]

```
release f;
```

p->?

```
*p = x;
```

p->?

binary semaphore 인지 안다고 가정 (f=0 || f=1)

관리자가 힙 셀을 가져가지 않음

RI_f : f=1 => p->? && [f=0 => emp]

여기서 관리자가 힙 셀을 가져가지 않음

```
acquire f when f > 0;
```

f=1 && p->?

```
  f = f - 1;
```

f=0 && p->?

```
release f;
```

p->?

```
*p = x;
```

p->?

binary semaphore 인지 안다고 가정 (f=0 || f=1)

엇갈린 두 이진 세마포어

RI_f : (f=1 => p->?) && (f=0 => emp)

RI_b : (b=1 => p->?) && (b=0 => emp)

```
while (true) {
  acquire f when f > 0;
    f = f - 1;
  release f;

  *p = x;

  acquire b when true;
    b = b + 1;
  release b;
}
```

||

```
while (true) {
  acquire b when b > 0;
    b = b - 1;
  release b;

  y = *p;

  acquire f when true;
    f = f + 1;
  release f;
}
```

이후 목표

- 구현 및 실험
- 복잡한 자료구조를 fine-grained lock 으로 다루는 알고리즘 검증
- 자원불변식 고정점에 재빠르게 도달하는 테크닉?
- 동시에 읽기 허용 등 확장 요소 고려