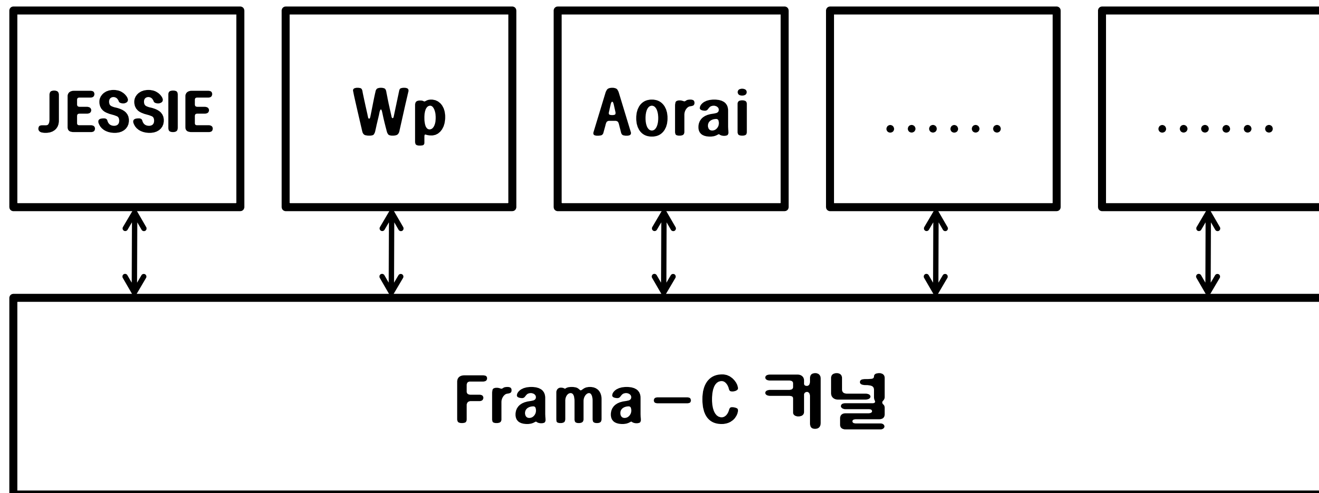
Frama-C 프로그램 검증 시스템 소개

박종현 @ POSTECH PL

Frama-C?

- **C 프로그램 대상 정적 분석 도구**
  - **플러그인** 구조



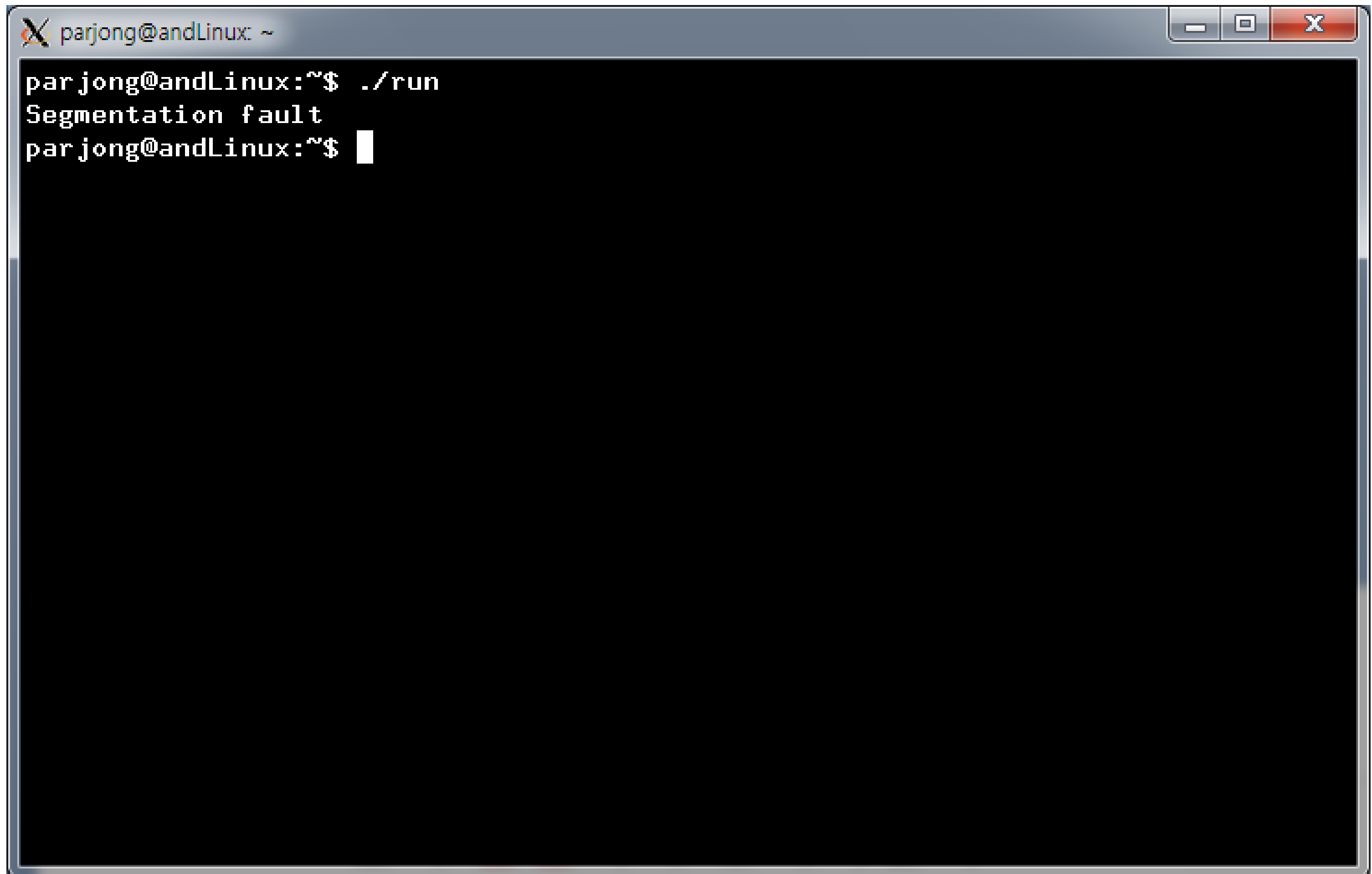
JESSIE?

- Frama-C **연역 검증** 플러그인
  - 프로그램 분석 ➔ 검증 조건 추출 ➔ 증명
  - **Hoare 논리**에 기초한 프로그램 검증 도구
- 사용법

  $ frama-c –jessie <검증 대상 C 파일>
- 특징
  - 다양한 자동 정리 증명기 및 증명 보조기 지원
  - 함수 단위로 안전성 검증 및 기능 검증 수행

안전성 검증?

- 프로그램이 **안전**하게 동작하는가?



안전성 검증?

- 프로그램이 **안전**하게 동작하는가?
  - 32비트 정수 표현 범위 안에서만 계산?
  - 할당된 메모리 영역만 참조?
  - 프로그램이 유한 시간 안에 종료?

- 예제

```
int mean(int p, int q) {
  return (p + q) / 2;
}
```

```
int mean(int p, int q) {
  return p / 2 + q / 2 + (p % 2 + q % 2) / 2;
}
```

기능 검증?

- 프로그램이 **의도한 기능**을 수행하는가?
  - sort 함수가 주어진 입력을 실제로 정렬하는가?

- 각 프로그램마다 의도하는 기능이 다름

```
void mean(int* p, int* q) ;
void sort(int* arr, int len);
```

- 사용자가 의도하는 기능에 대한 정보 필요

명세?

- 프로그램이 **의도하는 기능**에 대한 논리적 기술

```
/*@ 명세 */
int mean(int* p, int* q) {
  ......
}
```

- 종류?
  - **requires**
  - **ensures**
  - **assigns**
  - ......

requires & ensures

- requires P
  - 함수가 **호출**될 때 만족해야 하는 조건
  - \valid
- ensures P
  - 함수가 **종료**된 후 만족되어야 하는 조건
  - \result
- 예제

```
/*@ requires \valid(p) && \valid(q)
      ensures \result == (*p + *q) / 2; */
int mean(int* p, int* q) {
  return *p / 2 + *q / 2 + (*p % 2 + *q % 2) / 2;
}
```

requires & ensures

- ensures는 함수가 **종료**된 후 상태를 기술

```
/*@ requires \valid(p) && \valid(q);
      ensures \result == (*p + *q) / 2; */
int mean(int* p, int* q) {
  *p = 0; *q = 0; return 0;
}
```

- \old

```
/*@ requires \valid(p) && \valid(q);
      ensures \result == (\old(*p) + \old (*q)) / 2; */
int mean(int p, int q) {
   *p = 0; *q = 0; return 0;
}
```

requires & ensures

- ensures는 함수가 **종료**된 후 상태**만**을 기술

- 예제

```
/*@ requires \valid(p) && \valid(q);
      ensures \result == (\old(*p) + \old(*q)) / 2; */
int mean(int* p, int* q) {
  int i = 0;

  while( i >= 0 );

  return 0;
}
```

requires와 **ensures**만으로도 충분하다!

assigns

- 하지만…

```
int M;

/*@ requires \valid(p) && \valid(q);
      ensures \result == (\old(*p) + \old (*q)) / 2; */
int mean(int* p, int* q) {
  M = 0;
  return *p / 2 + *q / 2 + (*p % 2 + *q % 2) / 2;
}
```

- assigns L
  - 함수가 종료된 후 **변경될 수 있는 외부 메모리 영역** 한정
  - 기술되지 않은 메모리 영역은 **변경되지 않음**
  - \nothing

assigns

- 하지만...

```
int M;

/*@ requires \valid(p) && \valid(q);
      ensures \result == (\old(*p) + \old (*q)) / 2;
      assigns \nothing; */
int mean(int* p, int* q) {
  M = 0;
  return *p / 2 + *q / 2 + (*p % 2 + *q % 2) / 2;
}
```

- assigns L
  - 함수가 종료된 후 **변경될 수 있는 외부 메모리 영역** 한정
  - 기술되지 않은 메모리 영역은 **변경되지 않음**
  - \nothing

시연!

```
/*@ requires \valid(p) && \valid(q);
      ensures \valid == (\old(*p) + \old(*q)) / 2;
      assigns \nothing; */
int mean(int* p, int* q) {
  return *p / 2 + *q / 2 + (*p % 2 + *q % 2) / 2;
}
```

반복문!?

```
/*@ requires len >= 1 && \valid_range(arr, 0, len – 1);
      ensures 0 <= \result < len;
      ensures \forall int i; 0 <= i < len ==> arr[i] <= arr[\result];
      assigns \nothing; */
int find_max(int* arr, int len) {
  int idx = 0;

  for(int i = 1; i < len; i++)
    if( arr[i] > arr[idx] ) idx = i;

  return idx;
}
```

loop variant & loop invariant

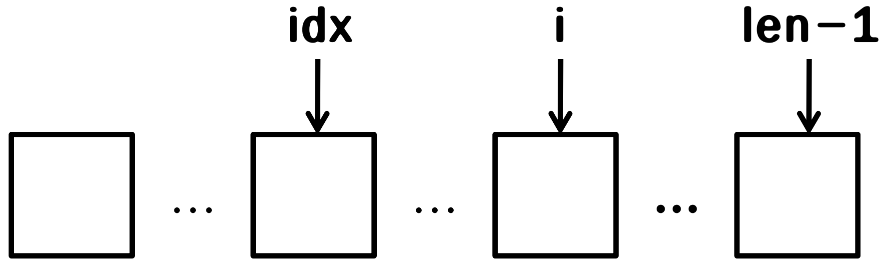
- loop variant e
  - 반복문 실행 시 값이 **줄어드는** 정수 계산식
- loop invariant P
  - 반복문 시작 점에서 항상 **만족되는** 조건

시연!

```
/*@ requires len >= 1 && \valid_range(arr, 0, len - 1);
      ensures 0 <= \result < len;
      ensures \forall int i; 0 <= i < len ==> arr[i] <= arr[\result];
      assigns \nothing; */
int find_max(int* arr, int len) {
  int idx = 0;

  /*@ loop variant len - i;
        loop invariant 0 <= i <= len;
        loop invariant 0 <= idx < i;
        loop invariant \forall int n; 0 <= n < i ==> arr[n] <= arr[idx]; */
  for(int i = 1; i < len; i++)
    if( arr[i] > arr[idx] ) idx = i;

  return idx;
}
```



포인터!?

```
typedef struct _list { int element; struct _list *next; } list;

/*@
      requires \valid(root) && ???;
      ensures ???;
      assigns \nothing;
 */
list *find_max(list *root) {
  list *ptr = root;

  while(root->next) {
    root = root->next;
    if( root->element > ptr->element ) ptr = root;
  }

  return ptr;
}
```

inductive & predicate

- inductive & predicate
  - 대상에 대한 논리적 성질 정의
- 예제

```
/*@ inductive reachable{L}(list *root, list *node) {
        case reachable_root {L}:
          \forall list* root; reachable(root, root);
        case reachable_next {L}:
          \forall list* root, *node;
            \valid(root) ==> reachable(root->next, node) ==>
            reachable(root, node);
      } */

//@ predicate finite_list{L}(list *l) = reachable(l, \null);
```

inductive & predicate

- 예제 (계속)

```
/*@ predicate connected_list{L}(list *root) =
        \forall list *node;
          reachable(root, node) ==>
          node == \null ||
          (\valid(node->next) && reachable(root, node->next)); */

//@ predicate linked_list{L}(list *l) = finite_list(l) && connected_list(l);
```

명세

```
/*@ requires \valid(root) && linked_list(root);
      ensures \valid(\result) && reachable(root, \result);
      ensures \forall list *node; \valid(node) ==> reachable(root, node) ==>
                                   node->element <= \result->element;
      assigns \nothing; */
list *find_max(list *root) {
  ......
}
```

```
/*@ requires len >= 1 && \valid_range(arr, 0, len – 1);
      ensures 0 <= \result < len;
      ensures \forall int i; 0 <= i < len ==> arr[i] <= arr[\result];
      assigns \nothing; */
```

명세 (계속)

loop variant???

```
list *ptr = root;

/*@ loop invariant \valid(root) && \valid(ptr);
      loop invariant reachable(\at(root, Pre), root);
      loop invariant reachable(\at(root, Pre), ptr);
      loop invariant \forall list *node;
        \valid(node) ==>
        reachable(\at(root, Pre), node) && !reachable(root->next, node) ==>
        node->element <= ptr->element; */
while(root
  root = r
  if( root-
}

return ptr;
```

```
/*@ loop variant len - i;
      loop invariant 0 <= i <= len;
      loop invariant 0 <= idx < i;
      loop invariant \forall int n; 0 <= n < i ==> arr[n] <= arr[idx]; */
```

logic

- logic
  - 입력에 따라 결과를 반환하는 논리 함수 정의
  - 프로그램 상의 값을 이용해야 할 경우?
    - 선언적 정의를 이용해서 함수 정의
- 예제

```
/*@ axiomatic Length {
      logic integer length{L}(list *root);
      axiom length_nil{L} : length(\null) == 0;
      axiom length_cons(L} : \forall list *l;
        \valid(l) ==> length(l) == length(l->next) + 1;
    } */

/*@ predicate finite_list{L}(list *l) =
      reachable(l, \null) &&
      \forall list *node; reachable(l, node) ==> length(node) >= 0; */
```

명세 (계속)

```
list *ptr = root;

/*@ loop variant length(root);
      loop invariant \valid(root) && \valid(ptr);
      loop invariant reachable(\at(root, Pre), ptr);
      loop invariant reachable(\at(root, Pre), root);
      loop invariant \forall list *node;
        \valid(node) ==>
        reachable(\at(root, Pre), node) && !reachable(root->next, node) ==>
        node->element <= ptr->element; */
while(root->next) {
  root = root->next;
  if( root->element > ptr->element ) ptr = root;
}

return ptr;
```

시연!

감사합니다!