

Modeling and Analysis of Mini-C Programs in Maude

목표 (Objective)

본 작업의 목표는 다음과 같다.

1. Mini-C 언어 정의

- 정수와 Bool을 기반으로 하는 단순화된 C 스타일 언어(Mini-C)를 정의한다.
- `+`, `-`, `*`, `/` 와 비교 연산(`<`, `>`, `<=`, `>=`), 논리 연산(`and`, `or`, `not`, `xor`)을 지원한다.
- `if`, `while`, `for`, `printf` 와 같은 대표적인 제어 구조와 출력 문장을 포함한다.

2. 힙/포인터 모델링 및 동적 메모리 연산

- 실행 상태에 **힙(Heap)** 개념을 도입하여, 정수 주소를 통한 동적 메모리 모델을 정의한다.
- C의 `malloc`, `free`, `p` (load), `p = v` (store)에 대응하는 연산을 언어 차원에서 설계하고, 힙 추상 자료구조와 연결되는 의미를 정의한다.

3. 메모리 누수 안전 속성 정의 및 분석

- 프로그램이 종료되었음에도 힙에 할당 블록이 남아 있는 상태를 "메모리 누수(leak)"로 정의한다.
- Maude의 `search` 기능을 이용하여, 주어진 Mini-C 프로그램이 leak 상태에 도달 가능한지 자동으로 탐색한다.
- `malloc` 후 `free` 를 수행하지 않는 경우, `for` 루프 내 반복 `malloc` 등 예제 프로그램에 대해 누수 여부를 분석한다.

이 목표를 통해, 단순 정수 프로그램(a-int program)에 머무르지 않고, **동적 메모리를 포함한 작은 C 스타일 언어**를 형식적으로 모델링하고, 안전 속성을 **Maude** 위에서 직접 검증할 수 있음을 보이는 것이 본 작업의 핵심이다.

설계 (Design)

전체 구조 개요

Mini-C 언어의 정의는 두 개의 Maude 모듈로 구성된다.

- 문법 모듈:** `fmod MINI-C-SYNTAX`
 - 정수/Bool, 식별자(Id), 식(Int/Bool), 문장 Stmt, 문장 리스트(StmtList) 등의 **구문(syntax)** 정의
 - `ifStmt`, `whileStmt`, `forStmt`, `mallocS`, `loadS`, `storeS`, `freeS` 등 C 스타일 문장 연산 정의
- 의미론 모듈:** `mod MINI-C`
 - 환경(Environment)** `Env`: 변수(Id) → 정수(Int) 매핑
 - 힙(Heap)**: 주소(Int) → 값(Int) 매핑
 - 출력(Out)**: 결과 출력 리스트
 - 구성(Config)**: `< Env , Heap , StmtList , Out >`
 - `eval`, `evalB`, `divInt` 등 식 의미론
 - `alloc`, `load`, `store`, `free` 등 힙 연산

- 각 문장을 한 단계씩 실행하는 재작성 규칙(rewrite rule)

이 구조를 통해, 언어의 문장 수준(예: `mallocS(x,1)`)과 힙 추상 연산(예: `alloc(H)`, `store(A,V,H)`)을 명확히 분리하였다.

Mini-C 문법 (MINI-C-SYNTAX)

문법 모듈에서는 다음과 같은 요소들을 정의한다.

- `Id` : 변수 이름 (예: `x`, `y`, `z`)
- `var : Id → Int` : 변수 참조를 Int 식으로 표현
- `Stmt`, `StmtList` : 문장과 문장 리스트
- `ifStmt`, `whileStmt`, `forStmt` : 제어 구조
- `printf` : 정수 출력
- `mallocS`, `loadS`, `storeS`, `freeS` : 동적 메모리 연산에 대응하는 문장

문장 리스트는 `...`와 `nil`로 이루어진 monoid 구조로 정의하여, C의 “세미콜론으로 이어지는 문장들”을 자연스럽게 표현한다.

상태 구성: Env / Heap / Config

의미론 모듈에서 프로그램 상태는 다음과 같이 정의한다.

- `Env` (Environment):
 - `emptyEnv` : 빈 환경
 - `X |→ N & E` : 식별자 `X`를 값 `N`에 매핑하고, 나머지 환경 `E`와 결합
 - `lookup(X, E)` : 환경 `E`에서 변수 `X`의 값을 조회 (`emptyEnv`에서는 0으로 가정)
 - `update(X, N, E)` : 환경 `E`에서 변수 `X`를 값 `N`으로 갱신
- `Heap` (힙):
 - `emptyHeap` : 빈 힙
 - `A |→ V & H` : 주소 `A`가 값 `V`를 저장하고, 나머지 힙 `H`와 결합
 - `alloc(H)` : 힙 `H`에서 최대 주소를 찾고, 그보다 1 큰 주소를 fresh 주소로 반환
 - `load(A, H)` : 주소 `A`의 값을 읽어옴 (존재하지 않는 경우 0)
 - `store(A, V, H)` : 주소 `A` 위치에 값 `V`를 기록
 - `free(A, H)` : 힙에서 주소 `A`에 해당하는 셀을 제거
- `Out` (출력):
 - `nilOut` : 비어 있는 출력
 - `N : O` : 앞에 정수 `N`을 붙인 출력 리스트
- 전체 구성 `Config` :

< Env , Heap , StmtList , Out >

- 초기 상태는

init(SL) = < emptyEnv , emptyHeap , SL , nilOut >

로 정의하여, 빈 환경과 빈 힙에서 주어진 프로그램 `SL` 을 실행하게 된다.

정수/Bool 식 의미론

- 정수식 평가 $eval : Int\ Env \rightarrow Int$
 - $eval(var(X), E) = lookup(X, E)$
 - $eval(A1 + A2, E) = eval(A1, E) + eval(A2, E)$
 - $eval(A1 - A2, E) = eval(A1, E) - eval(A2, E)$
 - $eval(A1 * A2, E) = eval(A1, E) * eval(A2, E)$
 - $eval(A1 / A2, E) = divInt(eval(A1, E), eval(A2, E))$
 - 정수 나눗셈은 별도 함수 `divInt` 로 정의하여, 0 이상, 양수 나눗셈에 대해서 재귀적으로 몫을 계산한다.
 - 그 외 패턴에 맞지 않는 Int 식(상수 등)은 $eq\ eval(A, E) = A\ [owise]$ 로 처리
 - Bool 식 평가 $evalB : Bool\ Env \rightarrow Bool$
 - 비교 연산: $A1 < A2$, $A1 > A2$, $A1 \leq A2$, $A1 \geq A2$ 는 각 $eval(A_i, E)$ 를 이용해 평가
 - 논리 연산: $not\ B$, $B1\ and\ B2$, $B1\ or\ B2$, $B1\ xor\ B2$ 를 재귀적으로 평가
 - 리터럴: $true$, $false$ 는 환경과 무관하게 그대로 반환
- 이를 통해 산술/논리 조건식을 모두 환경 기반으로 평가할 수 있다.

실행 규칙 (프로그램 한 단계 실행)

프로그램 실행은 구성 $\langle Env, Heap, StmtList, Out \rangle$ 에 대한 **재작성 규칙**으로 정의한다.

- 대입문 $X := A$:

```
rl [assign] :
  < E , H , (X := A) ; SL , O >
⇒
  < update(X, eval(A, E), E) , H , SL , O > .
```

- 출력 `printf(A)` :

```
rl [printf] :
  < E , H , printf(A) ; SL , O >
⇒
  < E , H , SL , eval(A, E) : O > .
```

- skip:

```
rl [skip] :
  < E , H , skip ; SL , O >
⇒
  < E , H , SL , O > .
```

- if문:

```

crl [if-true] :
  < E , H , ifStmt(B, SL1, SL2) ; SL , O >
⇒
  < E , H , SL1 ; SL , O >
  if evalB(B, E) == true .

crl [if-false] :
  < E , H , ifStmt(B, SL1, SL2) ; SL , O >
⇒
  < E , H , SL2 ; SL , O >
  if evalB(B, E) == false .

```

- **while**문은 **if** 기반으로 전개(desugaring):

```

rl [while] :
  < E , H , whileStmt(B, BODY) ; SL , O >
⇒
  < E , H ,
    ifStmt(B,
      BODY ; whileStmt(B, BODY) ; nil,
      skip ; nil) ;
    SL , O > .

```

- **for**문은 **Init ; while (Cond) { Body ; Step }** 으로 전개:

```

rl [for] :
  < E , H , forStmt(SL1, Bcond, SL2, BODY) ; SL , O >
⇒
  < E , H ,
    SL1 ;
    whileStmt(Bcond, BODY ; SL2) ;
    SL , O > .

```

동적 메모리 문장 의미론 (malloc / load / store / free)

문법에서 정의한 **mallocS**, **loadS**, **storeS**, **freeS** 는 힙 함수 **alloc**, **load**, **store**, **free** 를 호출하는 형태로 의미론이 정의된다.

- **X = malloc(n);**

```

rl [malloc] :
  < E , H , mallocS(X, N) ; SL , O >
⇒
  < update(X, alloc(H), E) ,
    store(alloc(H), 0, H) ,
    SL , O > .

```

- **alloc(H)** 로 fresh 주소를 할당하고,

- 해당 주소 위치를 `store` 를 통해 0으로 초기화한 후,
- 그 주소를 환경에서 변수 `x` 에 저장한다.

- `X = *p;`

```
rl [load] :
  < E, H, loadS(X, P) ; SL, O >
⇒
  < update(X, load(eval(P, E), H), E),
    H,
    SL, O > .
```

- 포인터 식 `P` 를 `eval` 로 평가한 주소에서,
- `load` 로 값을 읽어서 변수 `x` 에 저장한다.

- `p = v;`

```
rl [store] :
  < E, H, storeS(P, V) ; SL, O >
⇒
  < E,
    store(eval(P, E), eval(V, E), H),
    SL, O > .
```

- 포인터 식 `P` 와 값 식 `V` 를 평가하여, 해당 주소에 값을 저장한다.

- `free(p);`

```
rl [free] :
  < E, H, freeS(P) ; SL, O >
⇒
  < E,
    free(eval(P, E), H),
    SL, O > .
```

- 포인터 식 `P` 를 평가한 주소에 대해 `free` 를 호출하여, 해당 셀을 힙에서 제거한다.

이를 통해 C 스타일의 동적 메모리 연산을 언어 문장 수준과 힙 함수 수준으로 명확히 분리하여 모델링하였다.

메모리 누수 속성 정의 및 `search` 분석

- 메모리 누수(leak) 상태 정의
 - 프로그램 종료 상태:
 - 실행할 문장 리스트가 `nil` 인 상태 (`StmtList = nil`)
 - 힙에 남은 블록 존재:
 - 힙이 `emptyHeap` 이 아니고, 최소 한 개 이상의 셀이 존재하는 상태
 - 패턴으로는 `Heap = (A |⇒ V) & H` 꼴로 표현 가능

이를 종합하여, 메모리 누수 상태를 Maude 패턴으로 다음과 같이 정의한다.

< E:Env , (A:Int $\mid \Rightarrow$ V:Int) & H:Heap , nil , O:Out >

이 패턴에 도달 가능하다는 것은, “프로그램이 종료한 뒤에도 힙에 할당된 셀이 남아 있다” → “메모리 누수가 존재한다.”는 것을 의미한다.

- 예제 1 - `malloc` 만 하고 `free` 하지 않는 경우

```
search init(
  mallocS(x, 1) ;
  storeS(var(x), 99) ;
  nil
)
 $\Rightarrow$ * < E:Env , (A:Int  $\mid \Rightarrow$  V:Int) & H:Heap , nil , O:Out > .
```

- `Solution` 이 하나 이상 존재함
→ 종료 시 힙에 블록이 남아 있어 **누수가 발생**한다.

- 예제 2 - `malloc` 후 `free` 까지 하는 경우

```
search init(
  mallocS(x, 1) ;
  storeS(var(x), 99) ;
  freeS(var(x)) ;
  nil
)
 $\Rightarrow$ * < E:Env , (A:Int  $\mid \Rightarrow$  V:Int) & H:Heap , nil , O:Out > .
```

- 결과: `No solution.`
→ 종료 시 힙이 항상 비어 있어, **메모리 누수가 없다.**

- 예제 3 - `for` 루프에서 반복적으로 `malloc` 하는 경우

```
search init(
  (x := 0) ;
  forStmt(
    skip ; nil,
    var(x) < 3,
    (x := var(x) + 1) ; nil,
    mallocS(y, 1) ; nil
  ) ;
  nil
)
 $\Rightarrow$ * < E:Env , (A:Int  $\mid \Rightarrow$  V:Int) & H:Heap , nil , O:Out > .
```

- 루프가 3회 실행되며 서로 다른 주소에 할당이 누적되지만, `free` 가 없어 종료 상태에서 힙에 여러 블록이 남는다.
- 해당 패턴에 도달 가능한 `solution`이 존재함

- 루프 구조에서도 누수가 누적되는 것을 잡아낼 수 있음을 보여준다.

코드 (Implementation)

아래는 정리된 최종 Maude 코드이다.

- MINI-C-SYNTAX

```
fmod MINI-C-SYNTAX is
  protecting INT . --- Int, +, -, *, /, <, >, <=, >=
  protecting BOOL . --- Bool, true, false, and, or, xor, not

  --- 변수 이름
  sort Id .
  ops x y z : → Id [ctor] .

  --- 변수 참조: var(x)는 "x의 값"을 나타내는 Int 식
  op var : Id → Int [ctor] .

  --- 나눗셈 기호 (문법용)
  op _/_ : Int Int → Int [ctor] .

  --- 문장 / 문장 리스트
  sorts Stmt StmtList .
  subsort Stmt < StmtList .

  --- 문장 리스트 연산 (연속 실행)
  op nil : → StmtList [ctor] .
  op _;_ : StmtList StmtList → StmtList [ctor assoc id: nil] .
  --- 예: (x := 0) ; (y := 1) ; nil

  --- 기본 문장들
  op skip : → Stmt [ctor] .
  op _:=_ : Id Int → Stmt [ctor] . --- x := 3;

  --- if / while / for 문
  op ifStmt : Bool StmtList StmtList → Stmt [ctor] .
  op whileStmt : Bool StmtList → Stmt [ctor] .
  op forStmt : StmtList Bool StmtList StmtList → Stmt [ctor] .

  --- 출력
  op printf : Int → Stmt [ctor] .

  --- Heap 관련 문장들 (동적 메모리 연산)
  --- X = malloc(n);
  op mallocS : Id Int → Stmt [ctor] .

  --- X = *p; (p는 주소(포인터) 식)
```

```

op loadS : Id Int → Stmt [ctor] .

--- *p = v; (p: 주소 식, v: 값 식)
op storeS : Int Int → Stmt [ctor] .

--- free(p); (p: 주소 식)
op freeS : Int → Stmt [ctor] .

endfm

```

- 의미론 모듈: MINI-C

```

mod MINI-C is
  protecting MINI-C-SYNTAX .

  -----
  --- 1. 환경(Environment) : Id → Int
  -----

  sorts Env Binding .
  subsort Binding < Env .

  op emptyEnv : → Env [ctor] .
  op _&_ : Binding Env → Env [ctor] . --- (Binding & 나머지 환경)
  op _|→_ : Id Int → Binding [ctor] . --- x |→ 3

  --- lookup / update
  op lookup : Id Env → Int .
  op update : Id Int Env → Env .

  --- 선언 안 된 변수는 0이라고 가정
  vars X Y : Id .
  vars N M : Int .
  var E : Env .

  eq lookup(X, emptyEnv) = 0 .
  eq lookup(X, (X |→ N) & E) = N .
  ceq lookup(X, (Y |→ N) & E) = lookup(X, E)
    if X ≠ Y .

  eq update(X, N, emptyEnv) = (X |→ N) & emptyEnv .
  eq update(X, N, (X |→ M) & E) = (X |→ N) & E .
  ceq update(X, N, (Y |→ M) & E) = (Y |→ M) & update(X, N, E)
    if X ≠ Y .

  -----
  --- 2. 출력(Out)
  -----

  sort Out .

```

```

op nilOut : → Out [ctor] .
op _:_ : Int Out → Out [ctor] . --- 3 : 5 : nilOut

-----

--- 3. Heap : 주소(Int) -> 값(Int)
-----

sorts Heap HCell .
subsort HCell < Heap .

--- 빈 힙과 셀
op emptyHeap : → Heap [ctor] .
op _&_ : HCell Heap → Heap [ctor] . --- 셀 & 나머지 힙
op _|⇒_ : Int Int → HCell [ctor] . --- addr |⇒ value

--- Heap 연산
op maxAddr : Heap → Int . --- Heap 안에서 가장 큰 주소
op alloc : Heap → Int . --- fresh 주소
op load : Int Heap → Int . --- 메모리 읽기
op store : Int Int Heap → Heap . --- 메모리 쓰기
op free : Int Heap → Heap . --- 메모리 해제

vars A A' V W : Int .
var H : Heap .

eq maxAddr(emptyHeap) = 0 .
eq maxAddr((A |⇒ V) & H) =
  if A >= maxAddr(H) then A else maxAddr(H) fi .

eq alloc(H) = maxAddr(H) + 1 .

eq load(A, (A |⇒ V) & H) = V .
ceq load(A, (A' |⇒ V) & H) = load(A, H)
  if A =/= A' .
eq load(A, emptyHeap) = 0 . --- 없는 주소는 0으로 가정

eq store(A, V, emptyHeap) = (A |⇒ V) & emptyHeap .
eq store(A, V, (A |⇒ W) & H) = (A |⇒ V) & H .
ceq store(A, V, (A' |⇒ W) & H) = (A' |⇒ W) & store(A, V, H)
  if A =/= A' .

eq free(A, emptyHeap) = emptyHeap .
eq free(A, (A |⇒ V) & H) = H .
ceq free(A, (A' |⇒ V) & H) = (A' |⇒ V) & free(A, H)
  if A =/= A' .

-----

--- 4. 정수 / Bool 식 평가
-----

```

```

--- 정수식 평가: eval : Int(식) × Env → Int(값)
op eval : Int Env → Int .

vars A1 A2 Aint : Int .

eq eval(var(X), E) = lookup(X, E) .
eq eval(A1 + A2, E) = eval(A1, E) + eval(A2, E) .
eq eval(A1 - A2, E) = eval(A1, E) - eval(A2, E) .
eq eval(A1 * A2, E) = eval(A1, E) * eval(A2, E) .
eq eval(A1 / A2, E) = divInt(eval(A1, E), eval(A2, E)) .

```

```

--- 나머지 Int 식(상수 등)은 그대로 값으로 둬
eq eval(Aint, E) = Aint [owise] .

```

```

--- 정수 나눗셈 정의 (0 이상, 양수 나눗셈만)
op divInt : Int Int → Int .

```

```

vars I J : Int .
ceq divInt(I, J) = 0
  if I >= 0 and J > 0 and I < J .
ceq divInt(I, J) = divInt(I - J, J) + 1
  if I >= 0 and J > 0 and I >= J .

```

```

--- Bool 식 평가: evalB : Bool × Env → Bool
op evalB : Bool Env → Bool .

```

```

vars B B1 B2 : Bool .

```

```

eq evalB(A1 < A2, E) = (eval(A1, E) < eval(A2, E)) .
eq evalB(A1 > A2, E) = (eval(A1, E) > eval(A2, E)) .
eq evalB(A1 <= A2, E) = (eval(A1, E) <= eval(A2, E)) .
eq evalB(A1 >= A2, E) = (eval(A1, E) >= eval(A2, E)) .

```

```

--- 논리 연산
eq evalB(not B, E) = not evalB(B, E) .
eq evalB(B1 and B2, E) = evalB(B1, E) and evalB(B2, E) .
eq evalB(B1 or B2, E) = evalB(B1, E) or evalB(B2, E) .
eq evalB(B1 xor B2, E) = evalB(B1, E) xor evalB(B2, E) .

```

```

--- Bool 리터럴
eq evalB(true, E) = true .
eq evalB(false, E) = false .

```

```

-----
--- 5. 전체 구성 / 초기 상태
-----

```

```

sort Config .
op <_,_,_,_> : Env Heap StmtList Out → Config [ctor] .

```

```

--- 프로그램 시작 구성
op init : StmtList → Config .
eq init(SL) = < emptyEnv , emptyHeap , SL , nilOut > .

```

```

--- 규칙에서 사용할 변수들
vars SL SL1 SL2 BODY : StmtList .
var Bcond      : Bool .
var O          : Out .

```

```

-----
--- 6. 실행 규칙 (한 단계 실행)
-----

```

```

--- 대입문: X := A;
rl [assign] :
  < E , H , (X := A) ; SL , O >
⇒
  < update(X, eval(A, E), E) , H , SL , O > .

```

```

--- printf(A);
rl [printf] :
  < E , H , printf(A) ; SL , O >
⇒
  < E , H , SL , eval(A, E) : O > .

```

```

--- skip; S → S
rl [skip] :
  < E , H , skip ; SL , O >
⇒
  < E , H , SL , O > .

```

```

--- if 문
crl [if-true] :
  < E , H , ifStmt(B, SL1, SL2) ; SL , O >
⇒
  < E , H , SL1 ; SL , O >
if evalB(B, E) == true .

```

```

crl [if-false] :
  < E , H , ifStmt(B, SL1, SL2) ; SL , O >
⇒
  < E , H , SL2 ; SL , O >
if evalB(B, E) == false .

```

```

--- while B do BODY od
--- → if B then BODY ; while B do BODY od else skip fi
rl [while] :

```

```

    < E , H , whileStmt(B, BODY) ; SL , O >
⇒
    < E , H ,
      ifStmt(B,
        BODY ; whileStmt(B, BODY) ; nil,
        skip ; nil) ;
      SL , O > .

--- for (Init; Cond; Step) Body
--- ⇒ Init ; while (Cond) { Body ; Step }
rl [for] :
  < E , H , forStmt(SL1, Bcond, SL2, BODY) ; SL , O >
⇒
  < E , H ,
    SL1 ;
    whileStmt(Bcond, BODY ; SL2) ;
    SL , O > .

--- X = malloc(n);
rl [malloc] :
  < E , H , mallocS(X, N) ; SL , O >
⇒
  < update(X, alloc(H), E) ,
    store(alloc(H), 0, H) ,
    SL , O > .

--- X = *p;
rl [load] :
  < E , H , loadS(X, P) ; SL , O >
⇒
  < update(X, load(eval(P, E), H), E),
    H,
    SL , O > .

--- *p = v;
rl [store] :
  < E , H , storeS(P, V) ; SL , O >
⇒
  < E,
    store(eval(P, E), eval(V, E), H),
    SL , O > .

--- free(p);
rl [free] :
  < E , H , freeS(P) ; SL , O >
⇒
  < E,
    free(eval(P, E), H),

```

```
SL, O > .
```

```
endm
```