Towards Ownership Refinement Type Inference with Nested Arrays

Yusuke Fujiwara, Yusuke Matsushita, Kohei Suenaga and Atsushi Igarashi
Graduate School of Informatics, Kyoto University

The aim of this work

To extend **the ownership and refinement type system** ConSORT for imperative languages to support nested arrays.

ConSORT with Pointer Arithmetic [Tanaka et al., PEPM'24]

The automated verifier of the lack of assertion errors of low-level programs with integer array and pointer arithmetic

Reference type $(\lambda i.\tau)$ **ref** ^{r}

Type of pointers to array, whose elements are τ

Refinement type $\{\nu{:}\mathrm{int} \mid \varphi\}$

Integer type constrained by predicate φ

Ownership function *r*

- Function from array indices to **fractional ownership**
- Express read/write permissions for each reference
- r(i) is rational number satisfying 0 ≤r(i) ≤1
  - r(i)=1 : writable and readable, no other aliases are readable
  - 0<r(i)<1:readable only
- The sum of ownerships among aliases is no greater than 1.

Typing example

Program to initialize integer array

```
1  //init:⟨x : int, p : (λi.{ν : int | ⊤}) ref^[0,x−1]↦1⟩
   → ⟨x : int, p : (λi.{ν : int | ν = 0}) ref^[0,x−1]↦1 | int⟩
2  init (x, p){
3    if x = 0 then { 0 }
4    else {
5      p := 0;
6      //p:(λi.{ν : int | i = 0 ⟹ ν = 0}) ref^[0,x−1]↦1
7      let q = p + 1 in
8      //p:(λi.{ν : int | ν = 0}) ref^[0,0]↦1
9        q:(λi.{ν : int | ⊤}) ref^[0,x−2]↦1
10     init(x-1, q);
11     //p:(λi.{ν : int | ν = 0}) ref^[0,0]↦1
12       q:(λi.{ν : int | ν = 0}) ref^[0,x−2]↦1
13     alias(q = p + 1)
14     //p:(λi.{ν : int | ν = 0}) ref^[0,x−1]↦1
15   }
16 }
```

Overview

- Although their type system supports nested arrays, their implementation only supports unnested arrays. Furthermore, it is unclear how to extend their algorithm with nested arrays.

⇒ We formalize a type inference algorithm for ConSORT that supports nested arrays.

Reformalization of ownership *r*

$$r \quad ::= \quad (\varphi \mapsto q), r \quad | \quad \mathbf{true} \mapsto q,$$

- *r*:ownership term
- φ: a formula of the quantifier-free first-order logic with integer literals, variables, and symbols representing array indices
- q: a rational number satisfying $0 \le q \le 1$

Current status

- we have defined a constraint-generation algorithm that generates verification conditions
- from programs.
- We are currently implementing the type inference algorithm.
- The proof of soundness shown below is a future work.

Soundness

- A well-typed program will never experience an assertion failure.
- A well-typed program never gets stuck
  - No out-of-bound access occurs

Typing example of our work

- Int_array_init : A function that initializes the first length elements of the integer array arr to the integer num
- nested_array_init : A function that initializes the first length elements of an array of integer arrays arr
  - The integer arrays that are elements of arr have varying lengths and values depending on the index.

```
1  int_array_init (arr, length, num){
2    if length <= 0 then { 0 }
3    else {
4      arr := num;
5      let next_arr = arr + 1 in
6      let next_length = length - 1 in
7      let x = int_array_init(next_arr, next_length, num) in
8      alias(next_arr = arr + 1); 0
9    } }
10
11 // nested_array_init:
12 // <arr:(λi1.int ref) ref^(0≤i1∧i1≤length−1↦1), length:int> ->
13 // <arr:(λi1.(λi2.{ν:int | 0 ≤ i2∧i2 ≤ length−i1−1 ⟹ ν = length−i1})) ref^r2) ref^r1,
   length:int | int>
14 // r1 = (0 ≤ i1 ∧ i1 ≤ length − 1 ↦ 1)
15 // r2 = (0 ≤ i2 ∧ i2 ≤ length − i1 − 1 ↦ 1)
16 nested_array_init (arr, length){
17   if length <= 0 then { 0 }
18   else {
19     let sub_arr = alloc length : int ref in
20     let n = length in
21     let x = int_array_init(sub_arr, length, n) in
22     arr := sub_arr;
23     // arr:(λi1.(λi2.{ν:int | i1 = 0 ∧ 0 ≤ i2 ∧ i2 ≤ length − 1 ⟹ ν = n})) ref^r1) ref^r2
24     // r1 = (i1 = 0 ∧ 0 ≤ i2 ∧ i2 ≤ length − 1 ↦ 1), r2 = (0 ≤ i1 ∧ i1 ≤ length − 1 ↦ 1)
25     assert(**arr = n);
26     let next_length = length - 1 in
27     let next_arr = arr + 1 in
28     // arr:(λi1.(λi2.{ν:int | i1 = 0 ∧ 0 ≤ i2 ∧ i2 ≤ length − 1 ⟹ ν = n})) ref^r1) ref^r2
29     // r1 = (i1 = 0 ∧ 0 ≤ i2 ∧ i2 ≤ length − 1 ↦ 1), r2 = (i1 = 0 ↦ 1)
30     let y = nested_array_init (next_arr, next_length) in
31     alias(next_arr = arr + 1); 0
32     / arr:(λi1.(λi2.{ν:int | 0 ≤ i2∧i2 ≤ length−i1−1 ⟹ ν = length−i1})) ref^r2) ref^r1
33     // r1 = (0 ≤ i2 ∧ i2 ≤ length − i1 − 1 ↦ 1), r2 = (0 ≤ i1 ∧ i1 ≤ length − 1 ↦ 1)
34   } }
```