Precise Reasoning with Structured Heaps and Collective Operations *à la* Map/Reduce

Gregory Essertel, **Guannan Wei**, Tiark Rompf
Department of Computer Science, Purdue University

Motivating Example

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Many techniques failed to verify such properties in imperative programs with loops and dynamic allocations.

Because:

- Program abstractions are usually low-level *scalars*, rather than collections.
  e.g., a linked list contains natural numbers from `0` to `n-1`.
- Program abstractions lose the information of *time*.
  e.g., values at different loop iterations (loop context) are not distinguished.

DSL Can Help!

- We borrow ideas from Domain Specific Languages (DSLs):
  - Translate low-level imperative program `IMP` to high level functional program `FUN` with preserved semantics.

    `IMP` → `FUN`
- Introduce first-class *collective forms*:
  - The loop iteration index become an argument of the collective form instead of a free variable, thus the collective form is closed.
  - The value of collective form represents the value at the i_{th} loop.

    e.g., `λ(i).if (i > 0) then sum(i-1)+j(i-1) else 0`
- For the purpose of analyzing programs, we perform rule-based rewriting to obtain simplified collective forms.
- A simple arithmetic example:

```
IMP:
  int j = 0, sum = 0
  while (j < k) {
      sum = sum + j
      j = j + 1
  }
```

→

```
FUN:
  let j = λ(i).i + 1
  let sum = λ(i).(i+1)*i/2

STORE:
  j→k
  sum→(k − 1) * k / 2
```

Memory Allocations? Not a Problem

Consider the first loop in the motivating example:

- x and y is translated to address of object `ctx`. The object is tagged with the context (a unique program textual location), and indexed by the loop variable `i`.
- *newArray* is a primitive in language `FUN` that used to create arrays.
- The store represents the program state after running the first loop with n times.
- Then the second loop can use the store by dereferencing an address.
- ⊥ is uninitialized data.

ctx = root.snd.snd.while[i].fst

```
FUN:
  let y = λ(i).if (i>0) then &new:ctx[i]
                        else &new:ctx[0]
  let x = λ(i).if (i>0) then &new:ctx[i]
                        else &new:ctx[0]
  let ctx = λ(i).
    newArray(i₂ < i).
      [ head -> i₂,
        tail -> if (i₂>0) then &new:ctx[i₂-1]
                          else null ]

STORE:
  x→if (n>0) then &new:ctx[n-1] else null
  y→if (n>0) then &new:ctx[n-1] else ⊥
  ctx→if (n>0) then newArray(i < n).
          [ head -> i,
            tail -> if (i > 0) then
                      &new:ctx[i-1] else null ]
        else ⊥
```

Evaluation

- We implement a prototypical tool `SIGMA` for a subset of C.
- Evaluate against with CPAChecker and SeaHorn on SV-COMP benchmarks with loops. (Cells in red are incorrect result.)

Name	CPAchecker	SeaHorn	SIGMA
simple_built_from_end_true-unreach-call.i	TIMEOUT	250	273
list_addnat_false-unreach-call.i	2890	190	215
list_addnat_true-unreach-call.i	305560	170	215
loop_addnat_false-unreach-call.i	2830	190	285
loop_addnat_true-unreach-call.i	TIMEOUT	200	285
loop_addsubnat_false-unreach-call.i	3140	210	364
loop_addsubnat_true-unreach-call.i	TIMEOUT	230	364
nestedloop_mul1_true-unreach-call.i	OUT OF MEMORY	7280	405
nestedloop_mul2_true-unreach-call.i	TIMEOUT	240	365