An Effect System
Combining Alias and Liveness
for Explicit Memory Reuse

Oukseh Lee
ROPAS / KAIST
December 17, 2001
Garbage Collection

- Safe and efficient.
- Inefficient for frequent temporary heap allocations.

```haskell
fun incr l =
  case l of
    [] => []
  | h::t => (h+1)::(incr t)
```
Memory Reuse (or Recycle)

```haskell
fun incr l =
  case l of
    [] => []
  | h::t => (h+1)::(incr t) at l
```

- Operationally, destructive update.
- Cheap to evaluate.
Final Goal

Find an algorithm that inserts “safe and effective” memory recycle commands into ML programs.
Goal

Find a memory type system to guarantee the memory safety for ML with explicit memory recycle.
Memory Safety

\[
\begin{align*}
x &= 1 :: [] & x &= 1 :: [] \\
y &= 2 :: [] \text{ at } x & y &= 2 :: [] \text{ at } x \\
\text{case } x \text{ of } \ldots & & z &= 3 :: [] \text{ at } x
\end{align*}
\]

Conceptually, recycle = free + malloc.

We cannot recycle heap cells that will be used later (or recycled again).
$$
v ::= x \mid i \mid \lambda x.e
$$
$$
e ::= v
| \kappa \vec{v} \mid \text{case } v \ m \ldots \ m
| \text{let } x = e \ \text{in } e
| v \ v
| \kappa \vec{x} \ \text{at } x
$$
$$
m ::= \kappa \vec{x} \Rightarrow e
$$
Steps

How to abstract the heap?

- How to check the memory safety?
How to Abstract the Heap?

- Symbolic locations.
  - Names for sets of addresses.
- Two-level abstract domains.
  - Concrete.
  - Collapsed.
- Kinds of cells.
Symbolic Locations

- Names for sets of addresses.
- Allocations / recycles may introduce a new location.

\[
\begin{align*}
h::t & : \{X\} \\
x & : \{X\} \quad \frac{\text{h::t at } x::\{Y\}}{} \end{align*}
\]
Symbolic Locations for Functions

- Locations of arguments and locations newly allocated in the function are bound.

\[
\text{append} : \forall X, Y, Z. (X, Y) \rightarrow \{X, Y, Z\}
\]

- A function call is a substitution for bound locations.

\[
\text{append}(u, v) : \{U, V, W\}
\]
Two-Level Abstract Types

- Analyze the heap as concrete as possible.
- Need an abstraction for possibly infinite heap structure.

- Two-level abstractions:
  - Concrete: as concrete as the heap cells are.
  - Collapsed: ignoring the heap structure.
Concrete Type

\[
\langle \{A\}, \text{Node}, \left( \langle \{B\}, \text{Node}, \left( \langle \{C\}, \text{Leaf}, (\text{int}) \rangle, \right) \rangle, \langle \{D\}, \text{Leaf}, (\text{int}) \rangle \right), \langle \{E\}, \text{Leaf}, (\text{int}) \rangle \rangle
\]
Two-Level Abstract Types

Abstraction: concrete -> collapsed.
- When we join two different heap structures.

If ... then [] else [1,2]

Reconstruction: collapsed -> concrete.
- When we need the structure.

case x of h::t => ...

© Oukseh Lee, ROPAS/ KAIST
Collapsing the Structure!

\[ \{A, B, C, D, E\} \]
Cell-Kinds

- Diving a list into the spine and elements seems to be effective.

- Kinds of cells are its generalization.
  - Abstract data values: its constructor.
  - Tuples: its type.
Classifying by Cell-Kinds

\[
\begin{align*}
A & : \text{Node} \leftrightarrow \{A, B\} \\
    & : \text{Leaf} \leftrightarrow \{C, D, E\}
\end{align*}
\]
Two-Level Abstract Types

- Abstraction: concrete -> collapsed.
  - When we join two different heap structures.

  If ... then [] else [1,2]

- Reconstruction: collapsed -> concrete.
  - When we need the heap structure.

  case x of h::t => ...
How to Reconstruct?

\[
\begin{aligned}
\text{Node} & \leftrightarrow \{A, B\} \\
\text{Leaf} & \leftrightarrow \{C, D, E\}
\end{aligned}
\]
Naïve Reconstruction

\[
\left\langle \{A, B\}, \text{Node}, \left( \begin{array}{l}
\text{Node} \mapsto \{A, B\}, \text{Leaf} \mapsto \{C, D, E\}, \\
\text{Node} \mapsto \{A, B\}, \text{Leaf} \mapsto \{C, D, E\}
\end{array} \right) \right\rangle
\]
Accurate Reconstruction

\[
\left\langle \{F\}, \text{Node}, \left( \begin{array}{c}
\{\text{Node} \mapsto \{G\}, \text{Leaf} \mapsto \{I\}\}, \\
\{\text{Node} \mapsto \{H\}, \text{Leaf} \mapsto \{J\}\}
\end{array} \right) \right\rangle
\]
Then, Restore Locations

\[
\begin{align*}
\{A, B\}/F, & \{A, B\}/G, \{A, B\}/H, \\
\{C, D, E\}/I, & \{C, D, E\}/J
\end{align*}
\]
Sharing Flags

- If locations are shared, we do not use accurate reconstruction.
- **Sharing flag** for each set of locations.
Steps

- How to abstract the heap?

How to check the memory safety?
Liveness Analysis

- Collect used and recycled locations.

\[ \Delta \vdash e : \mu, U, R \]

- Used: by "case x of ...," or "1::[]."
- Recycled: by "at x."
Liveness Analysis

We cannot recycle locations that will be used later (or recycled again)!

\[
\begin{align*}
  e_1 & : U_1, R_1 \\
  e_2 & : U_2, R_2 \\
  R_1 \cap (U_2 \cup R_2) & = \emptyset
\end{align*}
\]
Function Calls Induce Aliases

- Between formal parameters.
- Between formal and actual parameters.

```ml
let val z = [1,2,3]
    fun f (x, y) = e
in  f (z, z)
end
```
Robust Against Aliases

If $A$ and $B$ may be aliased and $A$ is recycled, then $B$ must not be used (nor recycled).
Conclusion

- We present an effect system for checking every memory reuse is safe.
- Preliminary experiment encourages us:
  - GC overhead: 0-51.6% (in OCaml).
  - sieve runs 28.1% faster with reuse.
- Need to be proved.
- Need to devise an inference algorithm.
- Need to be extended to other language constructs.
Sharing Flags

- We can accurately reconstruct only non-shared collapsed-type.
Sharing Flags

- We can accurately reconstruct only non-shared collapsed-type.
Sharing Flags

- We can accurately reconstruct only non-shared collapsed-type.

\[
\begin{align*}
\{A, B\}/F, &\quad \{A, B\}/G, \quad \{A, B\}/H, \quad \{C, D\}/I, \quad \{C, D\}/J
\end{align*}
\]
Sharing Flags

- We can accurately reconstruct only non-shared collapsed-type.
Sharing Flags

- We can accurately reconstruct only non-shared collapsed-type.