An Effect System Combining Alias and Liveness for Explicit Memory Reuse (preliminary report)[*]

Oukseh Lee

Research On Program Analysis System[†]
Korea Advanced Institute of Science and Technology
cookcu@kaist.ac.kr

Abstract

The garbage collection is a safe and efficient method for managing the heap. However it is not efficient for temporary storages that are allocated often and deallocated quickly. Reusing temporary storages without collecting garbages can be a remedy for such inefficiency. We present an effect system for checking whether every explicit memory reuse is safe. We abstract the heap by using symbolic locations, kinds of cells, and two-level abstract domains. Based on this abstraction, we perform both alias and liveness analyses. The combined result precisely tells us whether every memory reuse is safe.

1 Introduction

1.1 Motivation

The garbage collection [7] is a safe and efficient method for managing the heap. In comparison with manual heap management, managing the heap with garbage collection does not make dangling pointers; the safety of the programs for accessing the heap is guarranted. Moreover, it was proved that the allocation and deallocation overhead of the garbage collection is cheaper than that of manual memory management [1]. Thus modern programming languages such SML [10] or Java [4] employ the garbage collection for their heap management.

One drawback of the garbage collection is that reusing heap cells always needs the garbage collection. Programs usually need heap cells for temporary values and those heap cells are discarded quickly. Because heap cells can be reused only by the garbage collection, frequent temporary heap allocations induce frequent garbage collections, which induce long execution time of programs.

Example 1 To illustrate temporary allocations induce frequent garbage collections, consider the following program:

[*]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology
[†]http://ropas.kaist.ac.kr

```
let fun incr x =
      case x of [] => []
              | h::t => (h+1)::(incr t)
in incr (incr (incr z))
end
```

Assume that the number of heap cells for `z` (integer list) is k and the heap size is greater than k. The above example allocates $3k$ heap cells because `incr` introduces a new list whose size is k. Thus if the heap size is less than $4k$ then the garbage collection occurs. Assume that only the result of the example will be accessed later. Then in function `incr`, without allocating new heap cells, we can just update an increased integer (`h+1`) into the cell for `h`. No new allocation means no garbage collection. □

As illustrated in Example 1, explicit memory reuse can be a remedy for such unexpected runtime overheads of the garbage collection. We believe that explicit memory reuse has lower runtime overhead than other compile-time garbage collection methods such as garbage marking and explicit deallocation [5].

1.2 Related Works

Many works have tried to help the garbage collection by program analyses however a cost-effective solution was not found yet.

The linear type systems [17, 8] are not accurate enough to help the garbage collection. The linear type system and its variants were attacking linear heap cells which can be deallocated after being used once. They are inaccurate for aliases and they are not practical because it needs extra runtime overhead for managing linear heap cells [6]. The alias type [12, 18] are more accurate than ordinary linear type, but it does not seem to be inferred automatically.

The region-based type system [15, 16] safely manages the heap by using regions which are collections of heap cells whose lifetimes are similar. Its accuracy and efficiency depends on how we partition heap cells into regions by region inference algorithms [13], however we cannot expect good partitioning for ordinary programs. Only region-friendly programs or programs with region annotated by programmers can run fast [14, 2, 3]. Moreover, regions are not adequate for recursive data structures such as lists or trees.

Shape aliases [11] are quite helpful for analyzing the memory reuse. Even though they focus on the heap structure and aliases, because their abstractions of the heap are quite sophisticated, they can be applied to the problem to help the garbage collection.

1.3 Our Work

We present a type system, combining alias and liveness analyses, for checking whether every memory reuse is safe. We abstract the heap by using symbolic locations, kinds of cells, and two-level abstract domains. Symbolic locations enables us to accurately analyze the heap even if heap cells are shared. The kind of cells is a generalization of the concept of the spine and elements in analyzing lists. Two-level abstract domains are for analyzing the heap structure in detail if necessary. By these facilities, we can estimate which heap cells are reachable from variables. Based on this abstraction, we perform both alias and liveness analyses. The combined result precisely tells us whether every memory reuse is safe.

2 Language

We use a monomorphic higher-order language:

$$
\begin{array}{rcll}
v & ::= & x \mid i \mid \lambda x.e & \text{variable, integer, function} \\
e & ::= & v & \text{value} \\
 & \mid & \kappa\,\vec{v} \mid \texttt{case}\ v\, m \cdots m & \text{data construction and case-expression} \\
 & \mid & \texttt{let}\ x \texttt{ = } e\ \texttt{in}\ e & \text{let-expression} \\
 & \mid & v\, v & \text{application} \\
 & \mid & \kappa\,\vec{v}\ \texttt{at}\ x & \text{memory reuse} \\
m & ::= & \kappa\,\vec{x} \Rightarrow e & \text{case branches}
\end{array}
$$

A vector denotes a sequence; that is, $\vec{v}$ denotes $(v_1, ..., v_n)$. Values are integers or functions. We assume that every variable in a program is different from each other. "$\kappa\,\vec{v}$" allocates a new heap cell and stores its constructor and elements to the heap cell. We assume that every data constructor has at least one argument and that the sizes for storing abstract data values are all the same. "`case` $v\, m \cdots m$" looks up the heap cell pointed by v and evaluates match $\kappa\,\vec{x} \Rightarrow e$ such that κ is equal to the data constructor in the heap cell. "`let` x `=` e_1 `in` e_2" binds x to the value that e_1 is evaluated to and evaluates e_2. "$v_1\, v_2$" is a function application that the function is v_1 and the argument is v_2. "$\kappa\,\vec{v}$ `at` x" is the same as "$\kappa\,\vec{v}$", but instead of allocating a new heap cell, it reuses the heap cell pointed by x. In order to simplify our memory-type system, we assume that functions cannot be stored into the heap. The exact semantics and the type system are in Appendix A and B, respectively.

3 A Memory-Type System

We abstract the heap using symbolic locations, cell-kinds, and two-level abstractions of the heap structure, and check whether every reuse is safe by performing both alias and liveness analyses. We describe those concepts in the memory-type system with the following example:

```
fun append (x, y) =
      case x of
        [] => y
      | h::t => (h :: append(t, y)) at x
```

Even though our language has no language constructor for tuples, for intuitive explanations, we assume that the type of function `append` is $(\langle \text{int}, \text{int}\rangle\ \text{list}, \langle \text{int}, \text{int}\rangle\ \text{list}) \rightarrow \langle \text{int}, \text{int}\rangle\ \text{list}$.

3.1 Symbolic Locations

Instead of using real addresses of the heap, we use symbolic locations as an abstraction of a set of addresses. For example, function `append(x,y)` takes two lists that start from `x` and `y`. We name the addresses of the list that starts from `x` (or `y`) as l_x (or l_y). Then function `append` gives us, as a result, an address that can reach l_x, l_y and l_{new} where l_{new} denotes the addresses of the heap cells newly allocated in evaluating the function body.

$$\texttt{append} : (l_x, l_y) \rightarrow \{l_x, l_y, l_{\text{new}}\} \tag{1}$$

Locations in a function type are polymorphic; they can be substituted by other locations when the function is called. For example, when `append([1],[2])` is called, the result is

$\{l_1, l_2, l_3\}$ where l_1 denotes the addresses of `[1]`, l_2 denotes the addresses of `[2]`, and l_3 denotes the addresses of heap cells newly allocated during `append([1],[2])`.

$$\texttt{append} : \forall l_x, l_y, l_{\text{new}}.(l_x, l_y) \to \{l_x, l_y, l_{\text{new}}\} \tag{2}$$

3.2 Cell-Kinds: Fine-Grained Types for Heap Cells

Dividing a list into its spine and elements seems to be effective for analyzing the heap [14]. For example, the description (2) of function `append` in Section 3.1 is not enough because it does not say the result of `append` includes only a part of l_x. An exact description for the `append` function is that it takes two lists `x` and `y`, gives us a list such that its elements consist of elements of both `x` and `y`, and its spine consists of `y`'s spine and newly allocated heap cells.

We propose a *cell-kind* as a generalization of spine and elements:

$$\mathit{CellKind} \quad k \ ::= \ \kappa \mid \langle \vec{\tau} \rangle$$

For abstract data values, the data constructors gives us finer classification of their heap cells than the types do. For tuples, we assume that their kinds are just their types.

By using cell-kinds, we can divide the spine and elements of a list. For example, a list whose type is $\langle \text{int}, \text{int} \rangle$ list can reach only $::$ cells and $\langle \text{int}, \text{int} \rangle$ cells with the assumption that `[]` is not stored in the heap. The $::$ cells are the spine and the $\langle \text{int}, \text{int} \rangle$ cells are the elements. Then we can write the memory-type of function `append` as:

$$\forall l_1, l_2, n_1, n_2, l_{\text{new}}. \left(\left\{ \begin{array}{l} :: \mapsto l_1 \\ \langle \text{int}, \text{int} \rangle \mapsto n_1 \end{array} \right\}, \left\{ \begin{array}{l} :: \mapsto l_2 \\ \langle \text{int}, \text{int} \rangle \mapsto n_2 \end{array} \right\} \right) \to \left\{ \begin{array}{l} :: \mapsto \{l_2, l_{\text{new}}\} \\ \langle \text{int}, \text{int} \rangle \mapsto \{n_1, n_2\} \end{array} \right\}.$$

It means `append` takes two arguments (both may reach $::$ cells and $\langle \text{int}, \text{int} \rangle$ cells) and gives us a result that may reach the elements ($\langle \text{int}, \text{int} \rangle$ cells) of the first argument (n_1), the spine and elements of the second argument (l_2 and n_2), a spine newly allocated during evaluating function `append`'s body (l_{new}).

3.3 Two-Level Abstractions of Heap Structure

We abstract the heap structure in two levels: a concrete memory-type and a collapsed memory-type. The concrete memory-type contains the information for the heap structure, but the collapsed memory-type does not.

$$\begin{array}{rrcl} \mathit{CollapsedType} & \bar{\mu} & \in & \mathit{CellKind} \to \wp(\mathit{Loc}) \times \mathit{Sharing} \\ \mathit{MemType} & \mu & ::= & \langle L, k, \vec{\mu} \rangle \mid \bar{\mu} \mid \cdots \end{array}$$

The concrete memory type $\langle L, k, \vec{\mu} \rangle$ describes immediately reachable heap cells; L is immediately reachable locations, k is its cell-kind, and $\vec{\mu}$ are the memory-types of its elements. The collapsed memory type is a map from each cell-kind to reachable locations of that kind. An additional sharing flag is for concretization which is explained in Section 3.5.

We abstract a concrete memory-type if necessary. We analyze the heap structure as concrete as possible, however in some cases, it is impossible to analyze the exact heap structure. For example, consider the following program:

```
if x=y then [] else [1].
```

The result of this expression can be either `[]` or a pointer to $::$ cell. Because the heap structure of these two results are different, we cannot represent the result as a concrete memory-type. In this case, collapsing the heap structure, we collect reachable locations classified by their cell-kinds. The memory-type of `[]` is $\emptyset$ and that of `[1]` is $\langle \{l\}, ::, (\mathrm{int}, \emptyset)\rangle$ which denotes a heap cell named as l which contains a list whose element is int type and whose tail is `[]`. Before joining these two memory-types, because their heap structures are different, we abstract them. The abstraction of $\emptyset$ is $\emptyset$ and the abstraction of $\langle \{l\}, ::, (\mathrm{int}, \emptyset)\rangle$ is a map $\{:: \mapsto \{l\}\}$. Thus the joined memory-type becomes $\{:: \mapsto \{l\}\}$ which denotes the result may reach $::$ cell named as l. In fact, the memory-type (2) of function `append` is based on the collapsed memory-type because we do not know the exact heap structure of the argument lists.

We also concretize a collapsed memory-type if necessary. In function `append`, we assumed the arguments have collapsed memory-types. In order to analyze the heap for the `case`-expression, we need to reconstruct the heap structure from the collapsed memory-type. We will explain how to reconstruct the heap structure in the end of Section 3.5.

Now we have an abstraction of the heap using symbolic locations, cell-kinds, and two-level abstract domains. By this model, we can estimate which heap cells are reachable from variables. However we cannot determine whether those heap cells can be safely reused yet.

3.4 Safe Reuse by Liveness and Aliases Analyses

Collecting used locations and reused locations enables us to check whether reused locations will not be used nor reused later. Used locations are the locations that is read or written by `case`-expressions or data constructions. Reused locations are the locations that are recycled by explicit memory reuse "`at` x." When e_2 are evaluated after evaluating e_1, we collect used and reused locations during evaluating e_1 and e_2. Then we can check whether the reused locations during evaluating e_1 is not used nor reused during evaluating e_2. In order to estimate used and reused locations for function calls, for each function memory-type, we have to keep the used and reused locations during evaluating the function body. Then function `append`'s memory type becomes:

$$\forall l_1, l_2, n_1, n_2, l_{\mathrm{new}}. \left(\left\{ \begin{array}{l} :: \mapsto l_1 \\ \langle \mathrm{int}, \mathrm{int}\rangle \mapsto n_1 \end{array} \right\}, \left\{ \begin{array}{l} :: \mapsto l_2 \\ \langle \mathrm{int}, \mathrm{int}\rangle \mapsto n_2 \end{array} \right\} \right) \overset{\{l_1\},\{l_1\}}{\rightarrow} \left\{ \begin{array}{l} :: \mapsto \{l_2, l_{\mathrm{new}}\} \\ \langle \mathrm{int}, \mathrm{int}\rangle \mapsto \{n_1, n_2\} \end{array} \right\}.$$

It means that function `append` uses the spine $(\{l_1\})$ of the first argument and reuses that spine $(\{l_1\})$.

The above liveness check is not enough for the memory safety because locations are not sound abstraction of real addresses. If two different locations denote the same addresses (two locations are aliased), it is fragile to check based on locations. For example, consider two locations l_1 and l_2. We conclude it is safe to use l_1 after reusing l_2 because they are different. However, it spoils the memory safety if two locations are aliased.

Whenever aliases are induced, we conservatively check whether the aliases spoil the memory safety. Instead of carrying aliases information, we analyze each expression with the assumption that different locations are not aliased. When aliases are induced by parameter passing, we check whether the function is robust against the aliases; we check whether every reuse in the function does not spoil the memory safety even under those aliases.

3.5 Typing Rules

The memory-type system is shown in Figure 1. A memory-type is an integer type, a concrete type, a collapsed type, or a function type. A concrete type consists of its possible locations, its cell-kind, and the memory-types of its elements. A collapsed type is a map from cell-kinds to possible reachable locations. For a collapsed type $\bar{\mu}$, $\bar{\mu}(\kappa) = \emptyset^1$ if $\kappa \notin \mathrm{dom}(\bar{\mu})$. Note that cell-kinds are only data constuctors because the language has no constructor for tuples. Each set of locations has a flag which denotes whether the locations have an shared structure (ω) or not (1). This sharing flag will be used only by concretization. A function type keeps used locations (U) and reused locations (R) during evaluating the function body. A (simultaneous) substitution S substitute locations by sets of locations. A symmetric alias relation A denotes whether two locations are aliased. typeof(κ) denotes the ordinary type of κ and typeof(x) denotes the type of x during typing it in the ordinary type system. kind(τ) denotes possible cell-kinds immediately reachable from a variable whose type is τ, and kinds(τ) denotes possible cell-kinds reachable from a variable whose type is τ.

A judgment "$\Delta \vdash v : \mu$" denotes value v has memory-type μ under memory-type environment Δ. An integer value has the integer type and a variable has the memory-type given by the memory-type environment.

$$\Delta \vdash i : \mathrm{int} \quad (\mathrm{int}) \qquad \Delta \vdash x : \Delta(x) \quad (\mathrm{var})$$

A function has a generalized function memory-type whose argument and result is adequate for the function body. The generalization is closing all the locations occurs in the function type except for free locations in Δ.

$$\frac{\Delta \cup \{x \mapsto \mu_1\} \vdash e : \mu_2, U, R}{\Delta \vdash \lambda x.e : \mathrm{Gen}_\Delta\left(\mu_1 \overset{U,R}{\to} \mu_2\right)} \ (\mathrm{fun})$$

A judgment "$\Delta \vdash e : \mu, U, R$" denotes that expression e has memory-type μ under memory-type environment Δ and that during evaluating e, the locations in U are used (read or written) and the locations in R are reused. A value expression does not use nor reuse anything.

$$\frac{\Delta \vdash v : \mu}{\Delta \vdash v : \mu, \emptyset, \emptyset} \ (\mathrm{value})$$

A data construction "$\kappa\vec{v}$" allocates a new heap cell and stores its constructor and elements to that cell. We name the new heap cell as l. The kind of the heap cell is κ, and its elements are the memory-type of v_i's. During constructing the data value, we use the heap cell ($\{l\}$).

$$\frac{\forall i.\Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\,\vec{v} : \langle\{l\}, \kappa, \vec{\mu}\rangle, \{l\}, \emptyset} \ (\mathrm{data})$$

A data construction "$\kappa\vec{v}$ `at` x" stores its constructor and elements to the heap cell pointed by x. We first looks up the memory-type of x. If it is a concrete memory-type, then we rename the heap cell (L) pointed by x as l, because we consider a reuse as a sequence of the deallocation of L and the allocation of l. Thus during this reuse, l is used and L is reused.

$$\frac{\Delta(x) = \langle L, \kappa', \vec{\mu}'\rangle \quad \forall i.\Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\,\vec{v}\ \mathtt{at}\ x : \langle\{l\}, \kappa, \vec{\mu}\rangle, \{l\}, L} \ (\mathrm{reuse})$$

$$\begin{array}{rcll}
\mathit{Loc} & l & & \\
 & L, U, R & \in & \wp(\mathit{Loc}) \\
\mathit{Sharing} & \pi & ::= & 1 \mid \omega \\
\mathit{DataConst} & \kappa & & \\
\mathit{CollapsedType} & \bar{\mu} & \in & \mathit{DataConst} \to \wp(\mathit{Loc}) \times \mathit{Sharing} \\
\mathit{MemType} & \mu & ::= & \mathrm{int} \mid \langle L, \kappa, \vec{\mu} \rangle \mid \bar{\mu} \mid \forall \vec{l}.\mu \mid \mu \overset{U,R}{\to} \mu \\
\mathit{MemTypeEnv} & \Delta & \in & \mathit{Var} \to \mathit{MemType} \\
\mathit{Subst} & S & \in & \mathit{Loc} \to \wp(\mathit{Loc}) \times \mathit{Sharing} \\
\mathit{Alias} & A & \in & \wp(\mathit{Loc} \times \mathit{Loc})
\end{array}$$

$\boxed{\Delta \vdash v : \mu}$

$$\Delta \vdash x : \Delta(x) \quad \text{(var)} \qquad \Delta \vdash i : \mathrm{int} \quad \text{(int)}$$

$$\frac{\Delta \cup \{x \mapsto \mu_1\} \vdash e : \mu_2, U, R}{\Delta \vdash \lambda x.e : \mathrm{Gen}_\Delta\left(\mu_1 \overset{U,R}{\to} \mu_2\right)} \text{(fun)}$$

$\boxed{\Delta \vdash e : \mu, U, R}$

$$\frac{\Delta \vdash v : \mu}{\Delta \vdash v : \mu, \emptyset, \emptyset} \text{(value)} \qquad \frac{\Delta(x) = \langle L, \kappa', \vec{\mu}' \rangle \quad \forall i. \Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\, \vec{v}\ \mathtt{at}\ x : \langle \{l\}, \kappa, \vec{\mu} \rangle, \{l\}, L} \text{(reuse)}$$

$$\frac{\forall i. \Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\, \vec{v} : \langle \{l\}, \kappa, \vec{\mu} \rangle, \{l\}, \emptyset} \text{(data)} \qquad \frac{\begin{array}{c}\Delta \vdash v_1 : \forall \vec{l}.(\mu_1 \overset{U,R}{\to} \mu_2) \quad \Delta \vdash v_2 : S\mu_1 \\ \mathrm{dom}(S) = \vec{l} \quad \vdash_P S : A \quad A \rhd \mu_1 \overset{U,R}{\to} \mu_2\end{array}}{\Delta \vdash v_1\, v_2 : S\mu_2, SU, SR} \text{(app)}$$

$$\frac{\begin{array}{l}\Delta \vdash y : \langle L, \kappa, \vec{\mu} \rangle \quad m_j = (\kappa\, \vec{x} \Rightarrow e) \\ \Delta \cup \{x_i \mapsto \mu_i\} \vdash e : \mu, U, R\end{array}}{\Delta \vdash \mathtt{case}\ y\ m_1 \cdots m_n : \mu, U \cup L, R} \text{(case)} \qquad \frac{\begin{array}{l}\Delta \vdash e_1 : \mu_1, U_1, R_1 \\ \Delta \cup \{x \mapsto \mu_1\} \vdash e_2 : \mu, U_2, R_2 \\ R_1 \cap (U_2 \cup R_2) = \emptyset\end{array}}{\Delta \vdash \mathtt{let}\ x \,\mathtt{=}\, e_1\ \mathtt{in}\ e_2 : \mu, U_1 \cup U_2, R_1 \cup R_2} \text{(let)}$$

$$\frac{\begin{array}{l}\forall \kappa \in \mathrm{kind}(\mathrm{typeof}(x)). \\ \quad (\mu'_\kappa, S_\kappa) = \gamma(\bar{\mu}, k) \\ \quad \Delta \cup \{x \mapsto \mu'_\kappa\} \vdash e : \mu_\kappa, U_\kappa, R_\kappa \\ \quad \vdash_C S_\kappa : A_\kappa \quad A_\kappa \rhd U_\kappa, R_\kappa \quad A_\kappa \rhd \mu_\kappa\end{array}}{\Delta \cup \{x \mapsto \bar{\mu}\} \vdash e : \bigsqcup S_\kappa \mu_\kappa, \bigcup S_\kappa U_\kappa, \bigcup S_\kappa R_\kappa} \text{(conc)}$$

Figure 1: The memory-type system.

Note that if the memory-type of x is not concrete, we concretize it by the (conc) rule prior to apply the (reuse) rule.

A `case`-expression "`case` $y\ m_1 \cdots m_n$" looks up the heap cell pointed by y and evaluates the match whose constructor is equal to that of the heap cell. We first look up the memory-type of y. If it is a concrete type, we can find the match whose constructor is equal to that of y's memory-type. We infer the memory-type of the body of the match binding each x_i to the elements of y's memory-type, and we add y's locations to used locations.

$$\frac{\begin{array}{l}\Delta \vdash y : \langle L, \kappa, \vec{\mu} \rangle \quad m_j = (\kappa\, \vec{x} \Rightarrow e) \\ \Delta \cup \{x_i \mapsto \mu_i\} \vdash e : \mu, U, R\end{array}}{\Delta \vdash \mathtt{case}\ y\ m_1 \cdots m_n : \mu, U \cup L, R} \text{(case)}$$

Similarly, if the memory-type of y is not concrete, we concretize it by the (conc) rule prior to

$\boxed{\vdash_P S : A}$

$$\begin{array}{ll}\vdash_P S : & \{l_1 \sim l_2 \mid (l_1 \mapsto L_1^{\pi_1}) \in S,\ (l_2 \mapsto L_2^{\pi_2}) \in S,\ L_1 \cap L_2 \neq \emptyset\} \cup \\ & \{l \sim l' \mid (l \mapsto L^{\pi}) \in S,\ l' \in L\}\end{array}$$

$\boxed{\vdash_C S : A}$

$$\vdash_C S : \ \{l \sim l' \mid (l \mapsto L^{\pi}) \in S,\ l' \in L\}$$

$\boxed{A \rhd \mu}$

$$\frac{\forall U, R \in \mu.\ A \rhd U, R}{A \rhd \mu}\ \text{(robust-type)}$$

$\boxed{A \rhd U, R}$

$$\frac{\forall (l_1 \sim l_2) \in A.\ l_1 \in R \ \text{ implies } \ l_2 \notin U \cup R}{A \rhd U, R}\ \text{(robust-UR)}$$

Figure 2: Definitions related to aliases.

apply the (case) rule.

In the case of `let`-expression, we check the condition that reused locations must not be used (nor reused). First we infer the memory-types of expressions e_1 and e_2. Because we evaluate e_2 after evaluating e_1, the locations reused during evaluating e_1 must not be used (nor reused) during evaluating e_2 ($R_1 \cap (U_2 \cup R_2) = \emptyset$).

$$\frac{\begin{array}{l}\Delta \vdash e_1 : \mu_1, U_1, R_1 \\ \Delta \cup \{x \mapsto \mu_1\} \vdash e_2 : \mu, U_2, R_2 \\ R_1 \cap (U_2 \cup R_2) = \emptyset\end{array}}{\Delta \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mu, U_1 \cup U_2, R_1 \cup R_2}\ \text{(let)}$$

In the case of a function call, we check the aliases induced by the function call spoil the memory safety. Parameter passing can be represented as a substitution. The domain of the substitution includes the formal parameters and the range of the substitution includes the actual parameters. Let S be a substitution that represents the parameter passing ($\Delta \vdash v_2 : S\mu_1$). In order that this function call does not spoil the memory safety, the function memory-type has to be robust against the aliases ($A \rhd \mu_1 \overset{U,R}{\to} \mu_2$) induced by the parameter passing ($\vdash_P S : A$).

$$\frac{\begin{array}{c}\Delta \vdash v_1 : \forall \vec{l}.(\mu_1 \overset{U,R}{\to} \mu_2) \quad \Delta \vdash v_2 : S\mu_1 \\ \mathrm{dom}(S) = \vec{l} \quad \vdash_P S : A \quad A \rhd \mu_1 \overset{U,R}{\to} \mu_2\end{array}}{\Delta \vdash v_1\ v_2 : S\mu_2, SU, SR}\ \text{(app)}$$

Aliases induced by parameter passing are (1) between parameters and (2) between parameters and free locations of the function. For formal parameters l_1 and l_2, if their actual parameters have an intersection, then l_1 and l_2 are aliased. A formal parameter l and the locations in its actual parameter L are aliased because the locations in the actual parameter may be free locations of the function.

$$\begin{array}{ll}\vdash_P S : & \{l_1 \sim l_2 \mid (l_1 \mapsto L_1^{\pi_1}) \in S,\ (l_2 \mapsto L_2^{\pi_2}) \in S,\ L_1 \cap L_2 \neq \emptyset\} \cup \\ & \{l \sim l' \mid (l \mapsto L^{\pi}) \in S,\ l' \in L\}\end{array}$$

A memory-type is robust against aliases if every pair of used and reused locations that occurs in the memory-type is robust against the aliases.

$$\frac{\forall U, R \in \mu.\ A \rhd U, R}{A \rhd \mu}\ \text{(robust-type)}$$

$$
\begin{array}{rcl}
1 \sqcup \pi & = & \pi \\
\omega \sqcup \pi & = & \omega \\
L_1^{\pi_1} \sqcup L_2^{\pi_2} & = & (L_1 \cup L_2)^{\pi_1 \sqcup \pi_2} \\
\mathrm{int} \sqcup \mathrm{int} & = & \mathrm{int} \\
\langle L_1, \kappa, \vec{\mu} \rangle \sqcup \langle L_2, \kappa, \vec{\mu}' \rangle & = & \langle L_1 \cup L_2, \kappa, (\mu_1 \sqcup \mu_1', \cdots, \mu_n \sqcup \mu_n') \rangle \\
\langle L_1, \kappa_1, \vec{\mu} \rangle \sqcup \langle L_2, \kappa_2, \vec{\mu}' \rangle & = & \alpha(\langle L_1, \kappa_1, \vec{\mu} \rangle) \sqcup \alpha(\langle L_2, \kappa_2, \vec{\mu}' \rangle) \text{ if } \kappa_1 \neq \kappa_2 \\
\langle L, \kappa, \vec{\mu} \rangle \sqcup \bar{\mu} & = & \alpha(\langle L, \kappa, \vec{\mu} \rangle) \sqcup \bar{\mu} \\
\bar{\mu} \sqcup \bar{\mu}' & = & \{\kappa \mapsto \bar{\mu}(\kappa) \sqcup \bar{\mu}'(\kappa) \mid \kappa \in \mathrm{dom}(\bar{\mu}) \cup \mathrm{dom}(\bar{\mu}')\} \\
\forall \vec{l}.\mu \sqcup \forall \vec{l}.\mu' & = & \forall \vec{l}.(\mu \sqcup \mu') \\
(\mu \overset{U_1,R_1}{\rightarrow} \mu_1) \sqcup (\mu \overset{U_2,R_2}{\rightarrow} \mu_2) & = & \mu \overset{U,R}{\rightarrow} (\mu_1 \sqcup \mu_2) \text{ where } U = U_1 \cup U_2 \text{ and } R = R_1 \cup R_2 \\
 & & \\
L_1^{\pi_1} \oplus L_2^{\pi_2} & = & \left\{ \begin{array}{ll} (L_1 \cup L_2)^{\pi_1 \sqcup \pi_2}, & \text{if } L_1 \cap L_2 = \emptyset \\ (L_1 \cup L_2)^{\omega}, & \text{if } L_1 \cap L_2 \neq \emptyset \end{array} \right. \\
\bar{\mu} \oplus \bar{\mu}' & = & \{\kappa \mapsto \bar{\mu}(\kappa) \oplus \bar{\mu}'(\kappa) \mid \kappa \in \mathrm{dom}(\bar{\mu}) \cup \mathrm{dom}(\bar{\mu}')\} \\
 & & \\
\alpha(\mathrm{int}) & = & \emptyset \\
\alpha(\bar{\mu}) & = & \bar{\mu} \\
\alpha(\langle L^{\pi}, \kappa, \vec{\mu} \rangle) & = & \{\kappa \mapsto L^{\pi}\} \oplus (\oplus \alpha(\mu_i)) \\
 & & \\
\gamma_{\kappa'}(\bar{\mu}) & = & (\langle L_0, \kappa', \vec{\mu} \rangle, \{l_{i\kappa} \mapsto \bar{\mu}(\kappa)\}) \\
 & & \quad \text{where typeof}(\kappa') = \vec{\tau} \rightarrow t \\
 & & \qquad\qquad L_0 = \{l_{0\kappa'}\} \text{ if } \bar{\mu}(\kappa') = L^1, \text{ or } L \text{ if } \bar{\mu}(\kappa') = L^{\omega} \\
 & & \qquad\qquad \mu_i = \{\kappa \mapsto \{l_{i\kappa}\}^1 \mid \kappa \in \mathrm{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^1\} \cup \\
 & & \qquad\qquad\qquad \{\kappa \mapsto \bar{\mu}(\kappa) \mid \kappa \in \mathrm{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^{\omega}\} \\
 & & \\
S(l) & = & \left\{ \begin{array}{ll} \{l\}^1 & \text{if } l \notin \mathrm{dom}(S) \\ S(l) & \text{if } l \in \mathrm{dom}(S) \end{array} \right. \\
S(L^{\pi}) & = & L_1^{\pi \sqcup \pi_1} \text{ where } L_1^{\pi_1} = \oplus_{l \in L} S(l) \\
S(L) & = & L_1 \text{ where } L_1^{\pi} = \oplus_{l \in L} S(l) \\
S(\mathrm{int}) & = & \mathrm{int} \\
S(\langle L, \kappa, \vec{\mu} \rangle) & = & \langle S(L), \kappa, (S\mu_1, \cdots, S\mu_n) \rangle \\
S(\bar{\mu}) & = & \{\kappa \mapsto S(L^{\pi}) \mid \kappa \mapsto L^{\pi} \in \bar{\mu}\} \\
S(\forall \vec{l}.\mu) & = & \forall \vec{l}.S\mu \text{ if } \mathrm{inv}(S) \cap \vec{l} = \emptyset \\
S(\mu_1 \overset{U,R}{\rightarrow} \mu_2) & = & S\mu_1 \overset{SU,SR}{\rightarrow} S\mu_2 \\
 & & \\
\mathrm{Gen}_{\Delta}(\mu) & = & \forall \vec{l}.\mu \text{ where } \vec{l} = \mathrm{free}(\mu) \backslash \mathrm{free}(\Delta)
\end{array}
$$

Figure 3: Definitions of join, abstraction, concretization, substitution, and generalization.

A pair of used and reused locations is robust against an alias $l_1 \sim l_2$ if the fact that l_1 is reused implies l_2 is not used nor reused.

$$
\frac{\forall (l_1 \sim l_2) \in A.\ l_1 \in R \text{ implies } l_2 \notin U \cup R}{A \rhd U, R} \ (\text{robust-UR})
$$

Note that for an alias $l_1 \sim l_2$, reusing l_1 and using l_2 do not always spoil the memory safety. It is safe to reuse l_1 *after* using l_2. However because the order of use and reuse is not kept, we conservatively conclude that reusing l_1 and using l_2 *may* spoil the memory safety.

Rule (conc) is for concretizing a collapsed memory-type. When we apply rule (case), if y does not have a concrete memory-type, then we cannot apply rule (case). After we concretize the memory-type of y, we can apply rule (case). The (conc) rule consists of two phases. One is concretizing the collapsed memory-type and inferring the memory-type of the expression with the concretized memory-type. The other is checking the result is robust against aliases induced by the concretization.

$$\frac{\begin{array}{l}\forall\kappa \in \mathrm{kind}(\mathrm{typeof}(x)).\\ \quad (\mu'_\kappa, S_\kappa) = \gamma_\kappa(\bar{\mu})\\ \quad \Delta \cup \{x \mapsto \mu'_\kappa\} \vdash e : \mu_\kappa, U_\kappa, R_\kappa\\ \quad \vdash_C S_\kappa : A_\kappa \quad A_\kappa \rhd U_\kappa, R_\kappa \quad A_\kappa \rhd \mu_\kappa\end{array}}{\Delta \cup \{x \mapsto \bar{\mu}\} \vdash e : \bigsqcup S_\kappa \mu_\kappa, \bigcup S_\kappa U_\kappa, \bigcup S_\kappa R_\kappa} \text{(conc)}$$

Concretizing is reconstructing the heap structure from a collapsed memory-type. For example, consider a list `::(1,::(2,[]))`. The concrete memory-type is:

$$\langle \{l_1\}, ::, (\mathrm{int}, \langle \{l_2\}, ::, (\mathrm{int}, \{\})\rangle)\rangle$$

where l_1 denotes the first `::` cell and l_2 denotes the second `::` cell. The abstraction of this concrete type is:

$$\{:: \mapsto \{l_1, l_2\}^1\}. \tag{3}$$

Now let us concretize the collapsed memory-type. A naively concretized one is:

$$\langle \{l_1, l_2\}, ::, (\mathrm{int}, \langle \{l_1, l_2\}, ::, (\mathrm{int}, \{\})\rangle)\rangle\,.$$

It is a safe concretization because it includes the original concrete type. However, it is an inaccurate concretization because it became to be invalid to use the second `::` cell ($\{l_1, l_2\}$) after reusing the first `::` cell ($\{l_1, l_2\}$).

Sharing flags help us to find an accurate concretization. Sharing flags denote whether locations are shared or not; a location is shared if there is more than one path to reach that location in a memory-type. If locations are not shared, then we can split them without any intersection between the partitions. For the above collapsed memory-type (3), the locations are not shared. Then the following concretization is possible.

$$\langle \{l'_1\}, ::, (\mathrm{int}, \langle \{l'_2\}, ::, (\mathrm{int}, \{\})\rangle)\rangle$$

where l'_1 and l'_2 are new locations. With this concretized memory-type, we infer the memory-type of the expression e. Later we confess that l'_1 and l'_2 are in fact $\{l_1, l_2\}$; we substitute l'_1 and l'_2 by $\{l_1, l_2\}$. Then the result type is the same as that of the naive concretization. The difference is that we can conclude that it is valid to use the second `::` cell ($\{l'_2\}$) after reusing the first `::` cell ($\{l'_1\}$) because they have no intersection. This concretization is safe only if locations are not shared. If $\{l_1, l_2\}$ has some shared locations, then it is impossible to split it to two partitions clearly.

Our concretization is defined as the follow:

$$\begin{array}{rcl}\gamma_{\kappa'}(\bar{\mu}) & = & (\langle L_0, \kappa', \vec{\mu}\rangle, \{l_{i\kappa} \mapsto \bar{\mu}(\kappa)\})\\ & & \quad \text{where } \mathrm{typeof}(\kappa') = \vec{\tau} \to t\\ & & \qquad L_0 = \{l_{0\kappa'}\} \text{ if } \bar{\mu}(\kappa') = L^1, \text{ or } L \text{ if } \bar{\mu}(\kappa') = L^\omega\\ & & \qquad \mu_i = \{\kappa \mapsto \{l_{i\kappa}\}^1 \mid \kappa \in \mathrm{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^1\} \cup\\ & & \qquad\qquad \{\kappa \mapsto \bar{\mu}(\kappa) \mid \kappa \in \mathrm{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^\omega\}\end{array}$$

The definition is sensitive to whether sets of locations are shared or not. If the set of locations are not shared, we introduce new locations and gives a substitution for restoring new locations to the original conservative ones. If the set of locations are shared, we do the same as the naive concretization. This concretization may induce aliases. Because the new locations will be substituted by the original locations, aliases are possible between new ones and original ones.

$$\vdash_C S : \{l \sim l' \mid (l \mapsto L^{\pi}) \in S,\ l' \in L\}$$

Thus the result must be robust against the induced aliases. The difference from aliases induced by parameter passing is that aliases are possible only between original ones and new ones. It is because locations are newly introduced only if they are not aliased.

3.6 Memory Safety

Similarly to the type soundness [19], we have to prove the memory safety which is expected for the memory-type system.

Conjecture 1 (Memory Safety) *For a program e, if $\emptyset \vdash e : \tau$ and $\emptyset \vdash e : \mu, U, R$ then e does not terminate or $(\{\}, \epsilon, e) \to^* (H, \epsilon, v)$.*

Only the memory-type system cannot guarantee a well-typed program does not go wrong because an ill-typed program in the ordinary type system can be well-typed in the memory type system. The memory-type system checks only whether every reuse is safe. Thus if a program is well-typed in both the ordinary type system and the memory-type system, then it does not go wrong.

4 Discussion

Preliminary experiments encourage our memory-type system. Some benchmark codes produced by Objective Caml native compiler [9], have 0–51.6% garbage collection overheads. Especially, the symbolic processing programs have high overheads. To a small benchmark program `sieve`, we insert reuse commands which is proved by our memory-type system. The result code run 28.1% faster than the original code does. It shows the possibility to utilize our memory-type system.

The memory-type system has to be proved and many extensions are possible. As well as the memory safety of the memory-type system, we have to prove the effectiveness of memory reuse. Moreover, in order to embed our memory-type system into real ML compilers, we need an inference algorithm that automatically inserts safe memory reuse commands to programs. We need to remove the restriction that functions cannot be stored at the heap. We need to extend our memory-type system with polymorphic types, reference values, closures, and continuations.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[2] David Gay and Alex Aiken. Memory management with explicit regions. In *Conference on Programming Language Design and Implementation*, June 1998.

[3] David Gay and Alex Aiken. Language support for regions. In *Conference on Programming Language Design and Implementation*, June 2001.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1997.

[5] G. W. Hamilton. Garbage recycling: Transforming programs to reuse garbage. Technical Report TR95-13, Department of Computer Science, University of Keele, Staffordshire, ST5 5BG, U. K., July 1995.

[6] Atsushi Igarashi and Naoki Kobayashi. Garbage collection based on linear type systems. In *Workshop on Types in Compilation*, 2000.

[7] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[8] Naoki Kobayashi. Quasi-linear types. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[9] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.02. Institut National de Recherche en Informatique et en Automatique, July 2001. `http://caml.inria.fr`.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[11] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[12] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.

[13] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.

[14] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report 98/25, Department of Computer Science, University of Copenhagen, 1998.

[15] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1994.

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[17] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *International Conference on Functional Programming and Computer Architecture*, June 1995.

[18] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, September 2000.

[19] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, 1992.

Appendix

A Semantics

$$\begin{array}{rrcl}
\mathit{Addr} & a & & \\
\mathit{Value} & u & ::= & i \mid a \mid \lambda x.e \\
\mathit{Cell} & c & ::= & \kappa\,\vec{u} \\
\mathit{Heap} & H & \in & \mathit{Addr} \to \mathit{Cell} \\
\mathit{Dump} & D & \in & (\mathit{Var} \times \mathit{Expr})^{\leq k} \\
\mathit{State} & & : & \mathit{Expr} \times \mathit{Heap} \times \mathit{Dump}
\end{array}$$

$$\begin{array}{lcll}
(\kappa\,\vec{u},\ H,\ D) & \to & (a,\ H \cup \{a \mapsto \kappa\,\vec{u}\},\ D) & \text{where } a \notin \mathrm{dom}(H) \\
(\kappa\,\vec{u}\ \texttt{at}\ a,\ H \cup \{a \mapsto c\},\ D) & \to & (a,\ H \cup \{a \mapsto \kappa\,\vec{u}\},\ D) & \\
(\texttt{case}\,a\,m_1 \cdots m_n,\ H,\ D) & \to & (e[u_i/x_i],\ H,\ D) & \text{if } H(a) = \kappa\,\vec{u} \text{ and } m_i = \kappa\,\vec{x} \Rightarrow e \\
(\texttt{let}\ x = e_1\ \texttt{in}\ e_2,\ H,\ D) & \to & (e_1,\ H,\ (x, e_2) \cdot D) & \\
((\lambda x.e)\ u,\ H,\ D) & \to & (e[u/x],\ H,\ D) & \\
(u,\ H,\ (x, e) \cdot D) & \to & (e[u/x],\ H,\ D) &
\end{array}$$

B Type System

$$\begin{array}{rrcl}
\mathit{Type} & \tau & ::= & \mathrm{int} \mid t \mid \vec{\tau} \to \tau \\
\mathit{TypEnv} & \Gamma & : & \mathit{Var} \to \mathit{Type}
\end{array}$$

$$\Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash i : \mathrm{int}$$

$$\frac{\Gamma + x{:}\tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \qquad \frac{\Gamma \vdash v_1 : \tau' \to \tau \quad \Gamma \vdash v_2 : \tau'}{\Gamma \vdash v_1\ v_2 : \tau}$$

$$\frac{\mathrm{typeof}(\kappa) = \vec{\tau} \to t \quad \forall i. \Gamma \vdash v_i : \tau_i}{\Gamma \vdash \kappa\,\vec{v} : t} \qquad \frac{\Gamma(x) = t' \quad \mathrm{typeof}(\kappa) = \vec{\tau} \to t \quad \forall i. \Gamma \vdash v_i : \tau_i}{\Gamma \vdash \kappa\,\vec{v}\ \texttt{at}\ x : t}$$

$$\frac{\Gamma \vdash v : t \quad \forall i. \Gamma, t \vdash m_i : \tau}{\Gamma \vdash \texttt{case}\ v\ m_1 \cdots m_n : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma + x{:}\tau' \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\mathrm{typeof}(\kappa) = \vec{\tau} \to t \quad \Gamma + x_i{:}\tau_i \vdash e : \tau}{\Gamma, t \vdash \kappa\,\vec{x} \Rightarrow e : \tau}$$