Inserting Safe Memory Reuse Commands into ML-like Programs*

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi

Dept. of Computer Science / Research On Program Analysis System**
Korea Advanced Institute of Science and Technology
{cookcu; hyang; kwang}@ropas.kaist.ac.kr

Abstract. We present a static analysis that estimates reusable memory cells and a source-level transformation that adds explicit memory-reuse commands into the program text. For benchmark ML programs, our analysis and transformation achieves the memory reuse ratio from 5.2% to 91.3%. The small-ratio cases are for programs that have too prevalent sharings among memory cells. For other cases, our experimental results are encouraging in terms of accuracy and cost. Major features of our analysis are: (1) poly-variant analysis of functions by parameterization for the argument heap cells; (2) use of multiset formulas in expressing the sharings and partitionings of heap cells; (3) deallocations conditioned by dynamic flags that are passed as extra arguments to functions; (4) individual heap cell as the granularity of explicit memory-free. Our analysis and transformation is fully automatic.

1 Overview

Our goal is to automatically insert explicit memory-reuse commands into ML-like programs so that they should not blindly request memory when constructing data.

We present a static analysis and a source-level transformation that adds explicit memory-reuse commands into the program text. The explicit memory-reuse is by inserting explicit memory-free commands right before data-construction expressions. Because the unit of both memory-free and allocation is an individual cell, such memory-free and allocation sequences can be implemented as memory reuses.[1]

Example 1. Function call "`insert i l`" returns a new list where integer `i` is inserted into its position in the sorted list `l`.

* This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

** `http://ropas.kaist.ac.kr`

[1] This approach's drawback might be that the memory reuse "bandwidth" is limited by the data-construction expressions in the program text. But our experimental results show that such a drawback is imaginary.

```
fun insert i l = case l of []   => i::[]                        (1)
                         | h::t => if i<h then i::l             (2)
                                   else h::(insert i t)         (3)
```

Let's assume that the argument list `l` is not used after a call to `insert`. If we program in C, we can destructively add one node for `i` into `l` so that the `insert` procedure should consume only one cons-cell. Meanwhile, the ML program's line `(3)` will allocate as many new cons-cells as that of the recursive calls. Knowing that list `l` is not used anymore, we can reuse the cons-cells from `l`:

```
fun insert i l = case l of []   => i::[]
                         | h::t => if i<h then i::l
                                   else let z = insert i t
                                        in  (free l; h::z)      (4)
```

In line `(4)`, "`free l`" will deallocate the single cons-cell pointed to by `l`. The very next expression's data construction "`::`" will reuse the freed cons-cell. □

1.1 Related Works

The type systems [25, 24, 2] based on linear logic fail to achieve Example 1 case because variable `l` is used twice. Kobayashi [10], and Aspinall and Hofmann [1] overcome this shortcoming by using more fine-grained usage aspects, but their systems still reject Example 1 because variable `l` and `t` are aliased at line `(2)`–`(3)`. They cannot properly handle aliasing: for "`let x=y in` e" where `y` points to a list, this list cannot in general be reused at e in their systems. Moreover, Aspinall and Hofmann did not consider an automatic transformation for reuse. Kobayashi provides an automatic transformation, but he requires the memory system to bookkeep a reference counter for every heap cell.

Deductive systems like the separation logic [9, 16, 17] and the alias-type system [18, 26] are powerful enough to reason about shared mutable data structures, but they cannot be used for our goal; they are not automatic. They need the programmer's help about memory invariants for loops or recursive functions.

The region-based memory managements [22, 23, 4, 5, 7] use a fixed partitioning strategy for recursive data structures, which is either implied by the programmer's region declarations or hard-wired inside the region-inference engine [20, 21]. Since every heap cell in a single region has the same lifetime, this "predetermined" partitioning can be too coarse; for example, transformations like the one in Example 1 are impossible.

Blanchet's escape analysis [3] and ours are both relational, covering the same class of relations (inclusion and sharing) among memory objects. The difference is the relation's targets and deallocation's granularity. His relation is between memory objects linked from program variables and their binding expression's results. Ours is between memory objects linked from any two program variables. His deallocation is at the end of a `let` or function body. Transformations like the one in Example 1 are impossible in his system. Harrison's [8] and Mohnen's [14] escape analyses have similar limitation: the deallocations is at the end of function body.

1.2 Our Solution

The features of our analysis and transformation are:

- In analyzing functions, parameterized abstract sharing-information between heap cells is maintained. The parameter in a function's analysis result is for the function's argument heap cells. A function's analysis result consists of terms called "*multiset formula.*" A multiset formula symbolically manifests an abstract sharing relation between heap cells.
- The parameterized multiset formula is instantiated at each function call, in order to finalize the sharing and/or disjointness properties for the function's input and output. This polyvariant analysis is not done by re-analyzing a function body multiple times.
- Partitioning of heap cells in a multiset formula is pivoted by two axes: one by structures (e.g. heads and tails for lists, roots and subtrees for trees, etc.) and the other by set exclusions (e.g. cells A excluding B). This double-axed partitioning is expressive enough to isolate proper reusable cells from others.
- Individual heap cell for each data constructor is the granularity of inserted memory-free commands.
- Dynamic flags are inserted to functions in order to condition their free commands on their call sites. Dynamic flags are simple boolean expressions composed of $\wedge$,$\vee$,and $\neg$.

Our contribution is a cost-effective automatic analysis and transformation for fine-grained memory reuses for recursive/algebraic data structures in ML-like programs. Our experimental results show that for small to large ML benchmark programs the memory reuse ratio ranges from 5.2% to 91.3%. The small-ratio cases expose that our analysis and transformation is weak for programs that have too prevalent sharings among memory cells. Other than those "torturing" cases, our experimental results are encouraging in terms of accuracy and cost. The analysis cost ranges from about 400 to 4500 lines per second. The limitation is that we only consider ML-like immutable recursive data.

Section 1.3 intuitively presents the features of our method for an example program. Section 2 defines the core of the target language, which consists of the source language plus explicit memory reuse commands. Section 3 presents the key abstract domain (memory-types) for our analysis. Section 4 shows, for the same example as in Section 1.3, a more detailed explanation on how our analysis and transformation works. Section 5 shows our experimental results and concludes.

1.3 Exclusion Among Heap Cells and Dynamic Flags

The accuracy of our algorithm depends on how precisely we can separate the two sets of heap cells: cells that are safe to deallocate and others that are not. If the separation is blurred, we hardly find deallocation opportunities.

For a precise separation of such two groups of heap cells, we have found that the standard partitioning by structures (e.g. heads and tails for lists, roots and subtrees for trees, etc.) is not enough. We need to refine the partitions by the notion of exclusion. Consider a function that builds a tree from an input tree. Let's assume that the input tree is not used after the call. In building the result tree, we want to reuse the nodes of the input tree. Can we free every node of the input? No, if the output tree shares some of its parts with the input tree. In that case, we can free only those nodes of the input that are *not* parts of the output. A concrete example is the following `copyleft` function. Both of its input and output are trees. The output tree's nodes along its left-most path are separate copies from the input tree and the rest are shared with the input tree.

```
fun copyleft t = case t of Leaf         => Leaf
                         | Node (t1,t2) => Node (copyleft t1, t2)
```

The `Leaf` and `Node` are the binary tree constructors. `Node` needs a heap cell that contains two fields to store the locations for the left and right subtrees. The opportunity of memory reuse is in the `case`-expression's second branch. When we construct the node after the recursive call, we can reuse the pattern-matched node of the input tree, but only when the node is *not* included in the output tree. Our analysis maintains such notion of exclusion.

Our transformation inserts `free` commands that are conditioned on dynamic flags passed as extra arguments to functions. These dynamic flags make different call sites to the same function have different deallocation behavior. By our `free`-commands insertion, above `copyleft` function is transformed to:

```
fun copyleft [β, β_ns] t =
  case t of Leaf         => Leaf
          | Node (t1,t2) => let p = copyleft [β ∧ β_ns, β_ns] t1
                            in  (free t when β; Node (p,t2))
```

Flag β is true when the argument `t` to `copyleft` can be freed inside the function. Hence the `free` command is conditioned on it: "`free t when` β." By the recursive calls, all the nodes along the left-most path of the input will be freed. The analysis with the notion of exclusion informs us that, in order for the `free` to be safe, the nodes must be excluded from the output. They are excluded if they are not reachable from the output. They are not reachable from the output if the input tree has no sharing between its nodes, because some parts (e.g. `t2`) of the input are included in the output. Hence the recursive call's actual flag for β is $\beta \wedge \beta_{\mathrm{ns}}$, where flag β_{ns} is true when there is no sharing inside the input tree.

1.4 Correctness Proof

The correctness of our analysis and transformation has been proved via a type system for safe memory deallocations [11]. We first proved our memory-type system sound: every well-typed program in the system does not access any deallocated heap cells. Then we proved that programs resulting from our analysis and transformation are always well-typed in the memory-type system. Since our transformation only inserts `free` commands, a transformed program's computational behavior modulo the memory-free operations remains intact.

Because of space limitation, we focus on our analysis and transformation in this paper. The details of our correctness proof are in [11].

SYNTAX

$$
\begin{array}{rll}
\textit{Type } \tau & ::= \mathsf{tree} \mid \mathsf{tree} \to \mathsf{tree} & \\
\textit{Boolean Expression } b & ::= \beta \mid \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b & \\
\textit{Storable Value } a & ::= \texttt{Leaf} \mid l & \\
\textit{Value } v & ::= a \mid x \mid \texttt{fix}\ x\ [\beta_1, \beta_2]\ \lambda x.e & \\
\textit{Expression } e & ::= v & \text{value} \\
& \mid\ \texttt{Node}\ (v, v) & \text{allocation} \\
& \mid\ \texttt{free}\ v\ \texttt{when}\ b & \text{deallocation} \\
& \mid\ \texttt{case}\ v\ (\texttt{Node}\ (x, y)\ \texttt{=>}\ e_1)\ (\texttt{Leaf}\ \texttt{=>}\ e_2) & \text{match} \\
& \mid\ v\ [b_1, b_2]\ v & \text{application} \\
& \mid\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e & \text{binding}
\end{array}
$$

OPERATIONAL SEMANTICS

$$
\begin{array}{rl}
h \in \mathsf{Heaps} & \stackrel{\Delta}{=} \mathsf{Locations} \stackrel{\text{fin}}{\to} \{(a_1, a_2) \mid a_i \text{ is a storable value}\} \\
f \in \mathsf{FreedLocations} & \stackrel{\Delta}{=} \wp(\mathsf{Locations}) \\
k \in \mathsf{Continuations} & \stackrel{\Delta}{=} \{(x_1, e_1) \ldots (x_n, e_n) \mid x_i \text{ is a variable and } e_i \text{ an expression}\}
\end{array}
$$

$$
\begin{array}{lll}
(\texttt{Node}\ (a_1, a_2), h, f, k) & \leadsto (l, h \cup \{l \mapsto (a_1, a_2)\}, f, k) & \\
\quad \text{where } l \text{ does not occur in } (\texttt{Node}\ (a_1, a_2), h, f, k) & & \\
(\texttt{free}\ l\ \texttt{when}\ b, h, f, k) & \leadsto (\texttt{Leaf}, h, f \cup \{l\}, k) & \text{if } b \Leftrightarrow \text{true},\ l \notin f,\ \text{and } l \in \text{dom}(h) \\
(\texttt{free}\ l\ \texttt{when}\ b, h, f, k) & \leadsto (\texttt{Leaf}, h, f, k) & \text{if } b \not\Leftrightarrow \text{true} \\
(\texttt{case}\ l\ (\texttt{Node}(x_1, x_2)\ \texttt{=>}\ e_1)\ (\texttt{Leaf}\ \texttt{=>}\ e_2), h, f, k) & \leadsto (e_1[a_1/x_1, a_2/x_2], h, f, k) & \\
\quad \text{where } h(l) = (a_1, a_2) \text{ and } l \notin f & & \\
(\texttt{case}\ \texttt{Leaf}\ (\texttt{Node}(x_1, x_2)\ \texttt{=>}\ e_1)\ (\texttt{Leaf}\ \texttt{=>}\ e_2), h, f, k) & \leadsto (e_2, h, f, k) & \\
((\texttt{fix}\ y\ [\beta_1, \beta_2]\ \lambda x.e)\ [b_1, b_2]\ v, h, f, k) & \leadsto & \\
\quad (e[(\texttt{fix}\ y\ [\beta_1, \beta_2]\ \lambda x.e)/y, b_1/\beta_1, b_2/\beta_2, v/x],\ h,\ f,\ k) & & \\
(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, h, f, k) & \leadsto (e_1, h, f, (x, e_2) \cdot k) & \\
(v, h, f, (x, e) \cdot k) & \leadsto (e[v/x], h, f, k) &
\end{array}
$$

Fig. 1. The Syntax and the Semantics.

2 Language

Figure 1 shows the syntax and semantics of the source language: a typed call-by-value language with first-order recursive functions, data constructions (memory allocations), de-constructions (case matches), and memory deallocations. All expressions are in the K-normal form [20, 10]: every non-value expression is bound to a variable by `let`. Each expression's value is either a tree or a function. A tree is implemented as linked cells in the heap memory. The heap consists of binary cells whose fields can store locations or a `Leaf` value. For instance, a tree `Node` (`Leaf`, `Node` (`Leaf`, `Leaf`)) is implemented in the heap by two binary cells l and l' such that l contains `Leaf` and l', and l' contains `Leaf` and `Leaf`.

The language has three constructs for the heap: $\texttt{Node}(v_1, v_2)$ allocates a node cell in the heap, and sets its contents by v_1 and v_2; a `case`-expression reads the contents of a cell; and $\texttt{free}\,v\,\texttt{when}\,b$ deallocates a cell v if b holds. A function has two kinds of parameters: one for boolean values and the other for an input tree. The boolean parameters are only used for the guards for `free` commands inside the function.

Throughout the paper, to simplify the presentation, we assume that all functions are closed, and we consider only well-typed programs in the usual monomorphic type system, with types being tree or $\mathsf{tree}{\rightarrow}\mathsf{tree}$. In our implementation, we handle higher-order functions, and arbitrary algebraic data types, not just binary trees. We explain more on this in Section 5.

The algorithm in this paper takes a program that does not have locations, `free` commands, or boolean expressions for the guards. Our analysis analyzes such programs, then automatically inserts the `free` commands and boolean parameters into the program.

3 Memory-Types: An Abstract Domain for Heap Objects

Our analysis and transformation uses what we call *memory-types* to estimate the heap objects for expressions' values. Memory-types are defined in terms of multiset formulas.

3.1 Multiset Formula

Multiset formulas are terms that allow us to abstractly reason about disjointness and sharing among heap locations. We call "multiset formulas" because formally speaking, their meanings (concretizations) are multisets of locations, where a shared location occurs multiple times.

The multiset formulas L express sharing configuration inside heap objects by the following grammar:

$$L ::= A \mid R \mid X \mid \pi.\mathsf{root} \mid \pi.\mathsf{left} \mid \pi.\mathsf{right} \mid \emptyset \mid L \mathbin{\dot{\sqcup}} L \mid L \mathbin{\dot{\oplus}} L \mid L \mathbin{\dot{\setminus}} L$$

Symbols A's, R's, X's and π's are just names for multisets of locations. A's symbolically denote the heap cells in the input tree of a function, X's the newly allocated heap cells, R's the heap cells in the result tree of a function, and π's for heap objects whose roots and left/right subtrees are respectively $\pi.\mathsf{root}$, $\pi.\mathsf{left}$, and $\pi.\mathsf{right}$. $\emptyset$ means the empty multiset, and symbol $\dot{\oplus}$ constructs a term for a multiset-union. The "maximum" operator symbol $\dot{\sqcup}$ constructs a term for the join of two multisets: term $L \mathbin{\dot{\sqcup}} L'$ means to include two occurrences of a location just if L or L' already means to include two occurrences of the same location. Term $L \mathbin{\dot{\setminus}} L'$ means multiset L excluding the locations included in L'.

Figure 2 shows the formal meaning of L in terms of abstract multisets: a function from locations to the lattice $\{0, 1, \infty\}$ ordered by $0 \sqsubseteq 1 \sqsubseteq \infty$. Note that we consider only good instantiations η of name X's, A's, and π's in Figure 2. The pre-order for L is:

$$L_1 \sqsubseteq L_2 \;\text{ iff }\; \forall\eta.\,\mathsf{goodEnv}(\eta) \Longrightarrow [\![L_1]\!]\eta \sqsubseteq [\![L_2]\!]\eta.$$

Semantics of Multiset Formulas

$$\begin{array}{lrl} \text{lattice} & \mathsf{Labels} & \triangleq \{0, 1, \infty\}, \text{ ordered by } 0 \sqsubseteq 1 \sqsubseteq \infty \\ \text{lattice} & \mathsf{MultiSets} & \triangleq \mathsf{Locations} \to \mathsf{Labels}, \text{ ordered pointwise} \end{array}$$

For all η mapping X's, A's, R's, $\pi.\mathsf{root}$'s, $\pi.\mathsf{left}$'s, and $\pi.\mathsf{right}$'s to $\mathsf{MultiSets}$,

$$\begin{array}{rll} [\![\emptyset]\!]\eta & \triangleq \bot & \\ [\![V]\!]\eta & \triangleq \eta(V) & (V \text{ is } X,\ A,\ R,\ \pi.\mathsf{root},\ \pi.\mathsf{left}, \text{ or } \pi.\mathsf{right}) \\ [\![L_1 \dot\sqcup L_2]\!]\eta & \triangleq [\![L_1]\!]\eta \sqcup [\![L_2]\!]\eta & \\ [\![L_1 \dot\oplus L_2]\!]\eta & \triangleq [\![L_1]\!]\eta \oplus [\![L_2]\!]\eta & \\ [\![L_1 \dot\setminus L_2]\!]\eta & \triangleq [\![L_1]\!]\eta \setminus [\![L_2]\!]\eta & \end{array}$$

where

$$\begin{array}{rl} \oplus \text{ and } \setminus & : \ \mathsf{MultiSets} \times \mathsf{MultiSets} \to \mathsf{MultiSets} \\ S_1 \oplus S_2 & \triangleq \lambda l.\, \text{if } S_1(l){=}S_2(l){=}1 \text{ then } \infty \text{ else } S_1(l) \sqcup S_2(l) \\ S_1 \setminus S_2 & \triangleq \lambda l.\, \text{if } S_2(l) = 0 \text{ then } S_1(l) \text{ else } 0 \end{array}$$

Requirements on Good Environments

$$\begin{array}{rl} \mathsf{goodEnv}(\eta) \triangleq & \text{for all different names } X \text{ and } X' \text{ and all } A, \\ & \quad \eta(X) \text{ is a } \textit{set} \text{ disjoint from both } \eta(X') \text{ and } \eta(A); \text{ and} \\ & \text{for all } \pi, \\ & \quad \eta(\pi.\mathsf{root}) \text{ is a } \textit{set} \text{ disjoint from both } \eta(\pi.\mathsf{left}) \text{ and } \eta(\pi.\mathsf{right}) \end{array}$$

Semantics of Memory-Types for Trees

$$[\![\langle L, \mu_1, \mu_2\rangle]\!]_{\mathsf{tree}}\, \eta \triangleq \{\langle l, h\rangle \mid h(l) = (a_1, a_2) \ \wedge\ [\![L]\!]\eta\, l \sqsupseteq 1 \ \wedge\ \langle a_i, h\rangle \in [\![\mu_i]\!]_{\mathsf{tree}}\, \eta\}$$

$$[\![L]\!]_{\mathsf{tree}}\, \eta \triangleq \left\{ \langle l, h\rangle \;\middle|\; \begin{array}{l} l \in \mathrm{dom}(h) \\ \wedge \forall l'.\, \mathsf{let}\ n = \text{number of different paths from } l \text{ to } l' \text{ in } h \\ \qquad \mathsf{in}\ (n \geq 1 \Rightarrow [\![L]\!]\eta\, l' \sqsupseteq 1) \ \wedge (n \geq 2 \Rightarrow [\![L]\!]\eta\, l' = \infty) \end{array} \right\}$$
$$\cup \{\langle \mathtt{Leaf}, h\rangle \mid h \text{ is a heap}\}$$

Fig. 2. The Semantics of Multiset Formulas and Memory-Types for Trees.

3.2 Memory-Types

Memory-types are in terms of the multiset formulas. We define memory-types μ_τ for value-type τ using multiset formulas:

$$\begin{array}{ll} \mu_{\mathsf{tree}} & ::= \langle L,\, \mu_{\mathsf{tree}},\, \mu_{\mathsf{tree}}\rangle \mid L \\ \mu_{\mathsf{tree}\to\mathsf{tree}} & ::= \forall A.A \to \exists X.(L, L) \end{array}$$

A memory-type μ_{tree} for a tree-typed value abstracts a set of heap objects. A heap object is a pair $\langle a, h\rangle$ of a storable value a and a heap h that contains all the reachable cells from a. Intuitively, it represents a tree reachable from a in h when a is a location; otherwise, it represents `Leaf`. A memory-type is either in a *structured* or *collapsed* form. A structured memory-type is a triple $\langle L,\, \mu_1,\, \mu_2\rangle$, and its meaning (concretization) is a set of heap objects $\langle l, h\rangle$ such that L, μ_1, and μ_2 abstract the location l and the left and right subtrees of $\langle l, h\rangle$, respectively. A collapsed memory-type is more abstract than a structured one. It is simply a multiset formula L, and its meaning (concretization) is a set of heap objects $\langle a, h\rangle$ such that L abstracts every reachable location and its sharing in $\langle a, h\rangle$. The formal meaning of memory-types is in Figure 2. The pre-order $\sqsubseteq_{\mathsf{tree}}$ for memory-types for trees is:

$$\mu_1 \sqsubseteq_{\mathsf{tree}} \mu_2 \;\text{ iff }\; \forall\eta.\mathsf{goodEnv}(\eta) \Longrightarrow [\![\mu_1]\!]_{\mathsf{tree}}\,\eta \subseteq [\![\mu_2]\!]_{\mathsf{tree}}\,\eta.$$

During our analysis, we switch between a structured memory-type and a collapsed memory-type. We can collapse a structured one by the $\mathsf{collapse}$ function:

$$\begin{aligned}\mathsf{collapse}(\langle L,\,\mu_1,\,\mu_2\rangle) &\stackrel{\Delta}{=} L \mathbin{\dot\sqcup} (\mathsf{collapse}(\mu_1) \mathbin{\dot\oplus} \mathsf{collapse}(\mu_2))\\ \mathsf{collapse}(\mu) &\stackrel{\Delta}{=} \mu \qquad\qquad\qquad\quad \text{(for collapsed } \mu)\end{aligned}$$

Note that when combining L and $\mathsf{collapse}(\mu_1) \mathbin{\dot\oplus} \mathsf{collapse}(\mu_2)$, we use $\dot\sqcup$ instead of $\dot\oplus$: it is because a root cell abstracted by L cannot be in the left or right subtree. We can also reconstruct a structured memory-type from a collapsed one when given splitting name π:

$$\begin{aligned}\mathsf{reconstruct}(L,\pi) &\stackrel{\Delta}{=} (\{\pi \mapsto L\}, \langle \pi.\mathsf{root}, \pi.\mathsf{left}, \pi.\mathsf{right}\rangle)\\ \mathsf{reconstruct}(\mu,\pi) &\stackrel{\Delta}{=} (\emptyset, \mu) \qquad\qquad\quad \text{(for structured } \mu)\end{aligned}$$

The second component of the result of $\mathsf{reconstruct}$ is a resulting structured memory-type and the first one is a record that L is a collection of $\pi.\mathsf{root}$, $\pi.\mathsf{left}$, and $\pi.\mathsf{right}$. The join of two memory-types is done by operator $\uplus$ that returns an upper-bound[2] of two memory-types. The operator $\uplus$ is defined using function $\mathsf{collapse}$:

$$\begin{aligned}L_1 \uplus L_2 &\stackrel{\Delta}{=} L_1 \mathbin{\dot\sqcup} L_2\\ \langle L,\,\mu_1,\,\mu_2\rangle \uplus \langle L',\,\mu_1',\,\mu_2'\rangle &\stackrel{\Delta}{=} \langle L \mathbin{\dot\sqcup} L',\,\mu_1 \uplus \mu_1',\,\mu_2 \uplus \mu_2'\rangle\\ L \uplus \langle L',\,\mu_1,\,\mu_2\rangle &\stackrel{\Delta}{=} L \mathbin{\dot\sqcup} \mathsf{collapse}(\langle L',\,\mu_1,\,\mu_2\rangle)\end{aligned}$$

For a function type $\mathsf{tree} \to \mathsf{tree}$, a memory-type describes the behavior of functions. It has the form of $\forall A.A \to \exists X.(L_1, L_2)$, which intuitively says that when the input tree has the memory type A, the function can only access locations in L_2 and its result must have a memory-type L_1. Note that the memory-type does not keep track of deallocated locations because the input programs for our analysis are assumed to have no `free` commands. The name A denotes all the heap cells reachable from an argument location, and X denotes all the heap cells newly allocated in a function. Since we assume every function is closed, the memory-type for functions is always closed. The pre-order for memory-types for functions is the pointwise order of its result part L_1 and L_2.

4 The `free`-Insertion Algorithm

We explain our analysis and transformation using the `copyleft` example in Section 1.3:

[2] The domain of memory-types for trees is not a lattice: the least upper-bound of two memory-types does not exist in general.

$$\begin{array}{rrcl}
\textit{Environment} & \Delta & \in & \{x \mid x \text{ is a variable}\} \xrightarrow{\text{fin}} \{\mu \mid \mu \text{ is a memory-type}\} \\
\textit{Bound} & \mathcal{B} & \in & \{V \mid V \text{ is } R \text{ or } \pi\} \xrightarrow{\text{fin}} \{L \mid L \text{ is a multiset formula}\} \\
\textit{Substitution} & \mathcal{S} & \in & \{L/V \mid V \text{ is } X \text{ or } A, \text{ and } L \text{ is a multiset formula}\}
\end{array}$$

$\boxed{\Delta \triangleright e : \mathcal{B}, \mu, L}$ Given environment Δ and expression e, we compute e's memory-type μ and usage L with a bound $\mathcal{B}$ for newly introduced R's and π's.

$$\frac{\Delta \triangleright v_1 : \mu_1 \quad \Delta \triangleright v_2 : \mu_2 \quad (\text{fresh } X)}{\Delta \triangleright \texttt{Node}\ (v_1, v_2) : \emptyset, \langle X,\ \mu_1,\ \mu_2\rangle, \emptyset}\ \text{U-NODE}$$

$$\frac{\Delta \triangleright v : \mu}{\Delta \triangleright v : \emptyset, \mu, \emptyset}\ \text{U-VALUE} \qquad \frac{\begin{array}{c}\Delta \triangleright e_1 : \mathcal{B}_1, \mu_1, L_1 \\ \Delta \cup \{x \mapsto \mu_1\} \triangleright e_2 : \mathcal{B}_2, \mu_2, L_2\end{array}}{\Delta \triangleright \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathcal{B}_1 \cup \mathcal{B}_2, \mu_2, L_1 \dot{\sqcup} L_2}\ \text{U-LET}$$

$$\frac{\begin{array}{l}(\mathcal{B}, \langle L,\ \mu_1',\ \mu_2'\rangle) \stackrel{\Delta}{=} \mathsf{reconstruct}(\mu, \pi) \quad (\text{fresh } \pi) \\ \Delta \cup \{x \mapsto \langle L,\ \mu_1',\ \mu_2'\rangle,\ x_1 \mapsto \mu_1',\ x_2 \mapsto \mu_2'\} \triangleright e_1 : \mathcal{B}_1, \mu_1, L_1 \\ \Delta \cup \{x \mapsto \emptyset\} \triangleright e_2 : \mathcal{B}_2, \mu_2, L_2\end{array}}{\begin{array}{r}\Delta \cup \{x \mapsto \mu\} \triangleright \texttt{case}\ x\ (\texttt{Node}\ (x_1, x_2)\ \texttt{=>}\ e_1)\ (\texttt{Leaf}\ \texttt{=>}\ e_2) : \\ \mathcal{B}_1 \cup \mathcal{B}_2 \cup \mathcal{B},\ \mu_1 \uplus \mu_2,\ L_1 \dot{\sqcup} L_2 \dot{\sqcup} L\end{array}}\ \text{U-CASE}$$

$$\frac{\begin{array}{l}\Delta \triangleright v_1 : \forall A.A \to \exists X.(L_1, L_2) \quad \Delta \triangleright v_2 : \mu_2 \\ \mathcal{S} \stackrel{\Delta}{=} [\mathsf{collapse}(\mu_2)/A,\ X'/X] \quad (\text{fresh } X', R)\end{array}}{\Delta \triangleright v_1\ v_2 : \{R \mapsto \mathcal{S}L_1\}, R, \mathcal{S}L_2}\ \text{U-APP}$$

$\boxed{\Delta \triangleright v : \mu}$ Given environment Δ and value v, we compute v's memory-type μ.

$$\frac{x \in \mathrm{dom}(\Delta)}{\Delta \triangleright x : \Delta(x)}\ \text{U-VAR} \qquad \frac{}{\Delta \triangleright \texttt{Leaf} : \emptyset}\ \text{U-LEAF}$$

$$\frac{\mu_{\text{lfp}} \stackrel{\Delta}{=} \text{fix}\left(\begin{array}{c}\lambda\mu.\ \forall A.A \to \exists X.(\mathsf{widen}_{\mathcal{B}}(\mathsf{collapse}(\mu')), \mathsf{widen}_{\mathcal{B}}(L)) \\ \text{where } \{f \mapsto \mu,\ x \mapsto A\} \triangleright e : \mathcal{B}, \mu', L\end{array}\right)}{\Delta \triangleright \texttt{fix}\ f\ \lambda x.e : \mu_{\text{lfp}}}\ \text{U-FUN}$$

Fig. 3. Step One: the Memory-Usage Analysis.

```
fun copyleft t = case t of Leaf          => Leaf                 (1)
                         | Node (t1,t2) => let p = copyleft t1   (2)
                                           in  Node (p,t2)       (3)
```

We first analyze the memory-usage of all expressions in the `copyleft` program; then, using the analysis result, we insert safe `free` commands to the program.

4.1 Step One: The Memory-Usage Analysis

Our memory-usage analysis (shown in Figure 3) computes memory-types for all expressions in `copyleft`. In particular, it gives the memory-type $\forall A.A \to \exists X.(A \dot{\sqcup} X,\ A)$ to `copyleft` itself. Intuitively, this memory-type says that when A denotes all the cells in the argument tree `t`, the application "`copyleft t`" may create new cells, named X in the memory-type, and returns a tree consisting of cells in A or X; but it uses only the cells in A.

This memory-type is obtained by a fixpoint iteration (U-FUN). We start from the least memory-type $\forall A.A \to \exists X.(\emptyset, \emptyset)$ for a function. Each iteration assumes that the recursive function itself has the memory-type obtained in the previous step, and the argument to the function has the (fixed) memory-type A. Under this assumption, we calculate the memory-type and the used cells for the function body. To guarantee the termination, the resulting memory-type and the used cells are approximated by "widening" after each iteration.

We focus on the last iteration step. This analysis step proceeds with five parameters A, X_2, X_3, X, and R, and with a splitting name π: A denotes the cells in the input tree `t`, X_2 and X_3 the newly allocated cells at lines `(2)` and `(3)`, respectively, X the set of all the newly allocated cells in `copyleft`, and R the cells in the returned tree from the recursive call "`copyleft t1`" at line `(2)`; the splitting name π is used for partitioning the input tree `t` to its root, left subtree, and right subtree. With these parameters, we analyze the `copyleft` function once more, and its result becomes stable, equal to the previous result $\forall A.A \to \exists X.(A \dot{\sqcup} X,\ A)$:

- **Line** `(1)`: The memory-type for `Leaf` is $\emptyset$, which says that the result tree is empty. (U-LEAF)
- **Line** `(2)`: The `Node`-branch is executed only when `t` is a non-empty tree. We exploit this fact to refine the memory-type A of `t`. We partition A into three parts: the root cell named $\pi.\mathsf{root}$, the left subtree named $\pi.\mathsf{left}$, and the right subtree named $\pi.\mathsf{right}$, and record that their collection is A: $\pi.\mathsf{root} \dot{\sqcup} (\pi.\mathsf{left} \dot{\oplus} \pi.\mathsf{right}) = A$. Then `t1` and `t2` have $\pi.\mathsf{left}$ and $\pi.\mathsf{right}$, respectively. (U-CASE)
  The next step is to compute a memory-type of the recursive call "`copyleft t1`." In the previous iteration's memory-type $\forall A.\,A \to \exists X.(A \dot{\sqcup} X,\ A)$ of `copyleft`, we instantiate A by the memory-type $\pi.\mathsf{left}$ of the argument `t1`, and X by the name X_2 for the newly allocated cells at line `(2)`. The instantiated memory-type $\pi.\mathsf{left} \to (\pi.\mathsf{left} \dot{\sqcup} X_2,\ \pi.\mathsf{left})$ says that when applied to the left subtree `t1` of `t`, the function returns a tree consisting of new cells or the cells already in the left subtree `t1`, but uses only the cells in the left subtree `t1`. So, the function call's result has the memory-type $\pi.\mathsf{left} \dot{\sqcup} X_2$, and uses the cells in $\pi.\mathsf{left}$. However, we use name R for the result of the function call, and record that R is included in $\pi.\mathsf{left} \dot{\sqcup} X_2$. (U-APP)
- **Line** `(3)`: While analyzing line `(2)`, we have computed the memory-types of `p` and `t2`, that is, R and $\pi.\mathsf{right}$, respectively. Therefore, "`Node (p,t2)`" has the memory-type $\langle X_3,\ R,\ \pi.\mathsf{right}\rangle$ where X_3 is a name for the newly allocated root cell at line `(3)`, R for the left subtree, and $\pi.\mathsf{right}$ for the right subtree. (U-NODE)

After analyzing the branches separately, we join the results from the branches. The memory-type for the `Leaf`-branch is $\emptyset$, and the memory-type for the `Node`-branch is $\langle X_3,\ R,\ \pi.\mathsf{right}\rangle$. We join these two memory-types by first collapsing $\langle X_3,\ R,\ \pi.\mathsf{right}\rangle$ to get $X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right})$, and then joining the two collapsed memory-types $X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right})$ and $\emptyset$. So, the function body has the memory-type $X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right})$.

Reduced Form $L_R ::= V \mid V \dot{\oplus} V \mid \emptyset \mid L_R \dot{\sqcup} L_R \qquad (V \text{ is } A \text{ or } X)$

$\boxed{\mathsf{widen}_{\mathcal{B}}(L)}$ gives a formula in a reduced form such that the formula only has free names A and X and is greater than or equal to L when $\mathcal{B}$ holds.

$$\mathsf{widen}_{\mathcal{B}}(L) \triangleq \mathcal{S}(\mathsf{reduce}_{\mathcal{B}}(L)) \qquad \text{(w1)}$$
$$(\mathcal{S} = \{X/X' \mid X' \text{ appears in } \mathsf{reduce}_{\mathcal{B}}(L)\} \text{ for the fixed } X)$$

where $\mathsf{reduce}_{\mathcal{B}}(L)$ uses the first available rule in the following:

$$\mathsf{reduce}_{\mathcal{B}}(R) \triangleq \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(R)) \qquad \text{(w2)}$$
$$\mathsf{reduce}_{\mathcal{B}}(\pi.o) \triangleq \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(\pi)) \qquad \text{(w3)}$$
$$\mathsf{reduce}_{\mathcal{B}}(L_1 \dot{\sqcup} L_2) \triangleq \mathsf{reduce}_{\mathcal{B}}(L_1) \dot{\sqcup} \mathsf{reduce}_{\mathcal{B}}(L_2) \qquad \text{(w4)}$$
$$\mathsf{reduce}_{\mathcal{B}}(L_1 \dot{\oplus} L_2) \triangleq \mathsf{reduce}_{\mathcal{B}}(L_1) \dot{\sqcup} \mathsf{reduce}_{\mathcal{B}}(L_2) \text{ (if } \mathsf{disjoint}_{\mathcal{B}}(L_1, L_2) \Leftrightarrow \text{true)} \qquad \text{(w5)}$$
$$(\mathsf{disjoint} \text{ is defined in Figure 6})$$
$$\mathsf{reduce}_{\mathcal{B}}(R \dot{\oplus} L) \triangleq \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(R) \dot{\oplus} L) \qquad \text{(w6)}$$
$$\mathsf{reduce}_{\mathcal{B}}(\pi.o_1 \dot{\oplus} \pi.o_2) \triangleq \begin{cases} \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(\pi) \dot{\oplus} \mathcal{B}(\pi)), & \text{if } o_1 = o_2 \\ \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(\pi)), & \text{otherwise} \end{cases} \qquad \text{(w7)}$$
$$\mathsf{reduce}_{\mathcal{B}}(\pi.o \dot{\oplus} L) \triangleq \mathsf{reduce}_{\mathcal{B}}(\mathcal{B}(\pi) \dot{\oplus} L) \qquad \text{(w8)}$$
$$\mathsf{reduce}_{\mathcal{B}}((L_1 \dot{\sqcup} L_2) \dot{\oplus} L_3) \triangleq \mathsf{reduce}_{\mathcal{B}}(L_1 \dot{\oplus} L_3) \dot{\sqcup} \mathsf{reduce}_{\mathcal{B}}(L_2 \dot{\oplus} L_3) \qquad \text{(w9)}$$
$$\mathsf{reduce}_{\mathcal{B}}((L_1 \dot{\oplus} L_2) \dot{\oplus} L_3) \triangleq$$
$$\mathsf{reduce}_{\mathcal{B}}(L_1 \dot{\oplus} L_2) \dot{\sqcup} \mathsf{reduce}_{\mathcal{B}}(L_2 \dot{\oplus} L_3) \dot{\sqcup} \mathsf{reduce}_{\mathcal{B}}(L_3 \dot{\oplus} L_1) \qquad \text{(w10)}$$
$$\mathsf{reduce}_{\mathcal{B}}(L) \triangleq L \qquad \text{(for all other } L) \text{ (w11)}$$

Fig. 4. The Widening Process.

How about the cells used by `copyleft`? In the `Node`-branch of the case-expression, the root cell $\pi.\mathsf{root}$ of the tree `t` is pattern-matched, and at the function call in line `(2)`, the left subtree cells $\pi.\mathsf{left}$ are used. Therefore, we conclude that `copyleft` uses the cells in $\pi.\mathsf{root} \dot{\sqcup} \pi.\mathsf{left}$.

The last step of each fixpoint iteration is widening: reducing all the multiset formulas into simpler yet more approximated ones. We widen the result memory-type $X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right})$ and the used cells $\pi.\mathsf{root} \dot{\sqcup} \pi.\mathsf{left}$ with the records $\mathcal{B}(R) = \pi.\mathsf{left} \dot{\sqcup} X_2$ and $\mathcal{B}(\pi) = A$. In the following, each widening step is annotated by the rule names of Figure 4:

$$\begin{array}{llr}
& X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right}) & \\
\sqsubseteq & X_3 \dot{\sqcup} ((\pi.\mathsf{left} \dot{\sqcup} X_2) \dot{\oplus} \pi.\mathsf{right}) & (\mathcal{B}(R) = \pi.\mathsf{left} \dot{\sqcup} X_2) \quad \text{(w6)} \\
= & X_3 \dot{\sqcup} (\pi.\mathsf{left} \dot{\oplus} \pi.\mathsf{right}) \dot{\sqcup} (X_2 \dot{\oplus} \pi.\mathsf{right}) & (\dot{\oplus} \text{ distributes over } \dot{\sqcup}) \quad \text{(w9)} \\
\sqsubseteq & X_3 \dot{\sqcup} A \dot{\sqcup} (X_2 \dot{\oplus} \pi.\mathsf{right}) & (\mathcal{B}(\pi) = A \text{ thus } \pi.\mathsf{left} \dot{\oplus} \pi.\mathsf{right} \sqsubseteq A) \quad \text{(w7)} \\
\sqsubseteq & X_3 \dot{\sqcup} A \dot{\sqcup} (X_2 \dot{\oplus} A) & (\mathcal{B}(\pi) = A \text{ thus } \pi.\mathsf{right} \sqsubseteq A) \quad \text{(w8)} \\
= & X_3 \dot{\sqcup} A \dot{\sqcup} X_2 \dot{\sqcup} A & (A \text{ and } X_2 \text{ are disjoint}) \quad \text{(w5)}
\end{array}$$

Finally, by replacing all the newly introduced X_i's by a fixed name X (w1) and by removing redundant A and X, we obtain $A \dot{\sqcup} X$. By rules (w4&w3) in Figure 4, $\pi.\mathsf{root} \dot{\sqcup} \pi.\mathsf{left}$ for the used cells is reduced to A.

The widening step ensures the termination of fixpoint iterations. It produces a memory-type all of whose multiset formulas are in a reduced form and can only have free names A and X. Note that there are only finitely many such multiset formulas that do not have a redundant sub-formula, such as A in $A \dot{\sqcup} A$. Consequently, after the widening step, only finitely many memory-types can be given to a function.

Although information is lost during the widening step, important properties of a function still remain. Suppose that the result of a function is given a multiset formula L after the widening step. If L does not contain the name A for the input tree, the result tree of the function cannot overlap with the input.[3] The presence of $\dot{\oplus}$ and A in L indicates whether the result tree has a shared sub-part. If neither $\dot{\oplus}$ nor A is present in L, the result tree can not have shared sub-parts, and if A is present but $\dot{\oplus}$ is not, the result tree can have a shared sub-part only when the input has.[4]

4.2 Step Two: `free` Commands Insertion

Using the result from the memory-usage analysis, our transformation algorithm (shown in Figure 5) inserts `free` commands, and adds boolean parameters β and β_{ns} (called *dynamic flags*) to each function. The dynamic flag β says that a cell in the argument tree can be safely deallocated, and β_{ns} that no sub-parts of the argument tree are shared. We have designed the transformation algorithm based on the following principles:

1. We insert `free` commands right before allocations because we intend to deallocate a heap cell only if it can be reused immediately after the deallocation.
2. We do not deallocate the cells in the result.

Our algorithm transforms the `copyleft` function as follows:

```
fun copyleft [β, βns] t =
  case t of Leaf         => Leaf                                (1)
          | Node (t1,t2) => let p = copyleft [β ∧ βns, βns] t1  (2)
                            in  (free t when β; Node (p,t2))    (3)
```

Note that "e_1 ; e_2" is an abbreviation of "`let` $x = e_1$ `in` e_2" when x does not appear in e_2.

The algorithm decides to pass $\beta \wedge \beta_{\mathrm{ns}}$ and β_{ns} in the recursive call (2). To find the first parameter, we collect constraints about conditions for which heap cells we should not free. Then, the candidate heap cells to deallocate must be disjoint with the cells to preserve. We derive such disjointness condition, expressed by a simple boolean expression. A preservation constraint has the conditional form $b \Rightarrow L$: when b holds, we should not free the cells in multiset L because, for instance, they have already been freed, or will be used later. For the first parameter, we get two constraints "$\neg\beta \Rightarrow A$" and "true $\Rightarrow X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right})$" from the algorithm in Figure 5 (rules I-FUN and I-LET). The first constraint

[3] This disjointness property of the input and the result is related to the usage aspects 2 and 3 of Aspinall and Hofmann [1].

[4] This sharing information is reminiscent of the "polymorphic uniqueness" in the Clean system [2].

Preservation Constraints $\quad \mathcal{C} \subseteq \{b \Rightarrow L \mid b \text{ is a boolean expression}\}$

$\boxed{\triangleright\, v_1^{(\Delta,\mu)} \Rightarrow v_2}$ takes v_1 annotated with the analysis result (Δ, μ), and produces free-inserted v_2.

$$\frac{}{\triangleright\, x \Rightarrow x}\ \text{I-VAR} \qquad \frac{}{\triangleright\, \texttt{Leaf} \Rightarrow \texttt{Leaf}}\ \text{I-LEAF} \qquad \frac{\mathcal{B}, \{\neg\beta \Rightarrow A\}, \text{true} \triangleright e \Rightarrow e' : \mathcal{C}}{\begin{array}{l}\triangleright\ \texttt{fix}\ f\ \lambda x.(e^{(\cdot,\mathcal{B},\cdot,\cdot)}) \\ \Rightarrow \texttt{fix}\ f\ [\beta, \beta_{\mathrm{ns}}] : \lambda x.e'\end{array}}\ \text{I-FUN}$$

$\boxed{\mathcal{B}, \mathcal{C}_1, b \triangleright e_1^{(\Delta,\mathcal{B}',\mu,L)} \Rightarrow e_2 : \mathcal{C}_2}$ takes an expression e_1 annotated with the analysis result $(\Delta, \mathcal{B}', \mu, L)$, a bound $\mathcal{B}$ for free names, and b and $\mathcal{C}_1$ that prohibit certain cells from being freed: b says that the result of e_1 should not be freed, and each $b' \Rightarrow L'$ in $\mathcal{C}_1$ that L' should not be freed when b' holds. The algorithm returns a `free`-inserted e_2 and $\mathcal{C}_2$ whose $b' \Rightarrow L'$ expresses that L' is freed in e_2 when b' holds.

$$\frac{\triangleright\, v \Rightarrow v'}{\mathcal{B}, \mathcal{C}, b \triangleright v \Rightarrow v' : \emptyset}\ \text{I-VALUE} \qquad \frac{\neg\exists x.\Delta(x) = \langle L, \mu_1, \mu_2\rangle \quad \triangleright\, v_1 \Rightarrow v_1' \quad \triangleright\, v_2 \Rightarrow v_2'}{\mathcal{B}, \mathcal{C}, b \triangleright (\texttt{Node}\ (v_1, v_2))^{(\Delta,\cdot,\cdot,\cdot)} \Rightarrow \texttt{Node}\ (v_1', v_2') : \emptyset}\ \text{I-NOF}$$

$$\frac{\begin{array}{c}\exists x.\Delta(x) = \langle L, \mu_1, \mu_2\rangle \quad \triangleright\, v_1 \Rightarrow v_1' \quad \triangleright\, v_2 \Rightarrow v_2' \\ \mathcal{C}' \triangleq \mathcal{C} \cup \{b \Rightarrow \mathsf{collapse}(\mu)\} \quad b' \triangleq \mathsf{free}_{\mathcal{B},\mathcal{C}'}(L)\end{array}}{\mathcal{B}, \mathcal{C}, b \triangleright (\texttt{Node}\ (v_1, v_2))^{(\Delta,\cdot,\mu,\cdot)} \Rightarrow (\texttt{free}\ x\ \texttt{when}\ b';\ \texttt{Node}\ (v_1', v_2')) : \{b' \Rightarrow L\}}\ \text{I-FREE}$$

$$\frac{\mathcal{B}, \mathcal{C}, b \triangleright e_1 \Rightarrow e_1' : \mathcal{C}_1 \quad \mathcal{B}, \mathcal{C}, b \triangleright e_2 \Rightarrow e_2' : \mathcal{C}_2}{\begin{array}{l}\mathcal{B}, \mathcal{C}, b\ \triangleright\ \texttt{case}\ x\ (\texttt{Node}\ (x_1, x_2)\ \texttt{=>}\ e_1)\ (\texttt{Leaf =>}\ e_2) \\ \qquad \Rightarrow \texttt{case}\ x\ (\texttt{Node}\ (x_1, x_2)\ \texttt{=>}\ e_1')\ (\texttt{Leaf =>}\ e_2') : \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}\ \text{I-CASE}$$

$$\frac{\begin{array}{l}\mathcal{B}, \mathcal{C} \cup \{\text{true} \Rightarrow L,\ b \Rightarrow \mathsf{collapse}(\mu)\}, \text{false} \triangleright e_1 \Rightarrow e_1' : \mathcal{C}_1 \\ \mathcal{B}, \mathcal{C} \cup \mathcal{C}_1, b \triangleright e_2 \Rightarrow e_2' : \mathcal{C}_2\end{array}}{\mathcal{B}, \mathcal{C}, b \triangleright \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ (e_2^{(\cdot,\cdot,\mu,L)}) \Rightarrow \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2' : \mathcal{C}_1 \cup \mathcal{C}_2}\ \text{I-LET}$$

$$\frac{\triangleright\, v \Rightarrow v' \quad L \triangleq \mathsf{collapse}(\mu) \quad b \triangleq \mathsf{free}_{\mathcal{B},\mathcal{C}}(L \mathbin{\dot{\setminus}} R) \quad b_{\mathrm{ns}} \triangleq \mathsf{nosharing}_{\mathcal{B}}(L)}{\mathcal{B}, \mathcal{C}, b'\ \triangleright\ (x\ (v^{(\Delta,\mu)}))^{(\cdot,\cdot,R,\cdot)} \Rightarrow x\ [b, b_{ns}]\ v' : \left\{b \hookrightarrow L \mathbin{\dot{\setminus}} R\right\}}\ \text{I-APP}$$

$\boxed{\mathsf{free}_{\mathcal{B},\mathcal{C}}(L)}$ calculates a safe condition to free L from the bound $\mathcal{B}$ for free names and the constraint $\mathcal{C}$ that says when certain cells should not be freed.

$$\mathsf{free}_{\mathcal{B},\mathcal{C}}(L) \triangleq \bigwedge \left\{\neg b \vee \mathsf{disjoint}_{\mathcal{B}}(L, L') \;\middle|\; (b \Rightarrow L') \in \mathcal{C}\right\}$$

Fig. 5. Step Two: the Algorithm to Insert `free` Commands.

means that we should not free the cells in the argument tree `t` if β is false, and the second that we should not free the cells in the result tree of the `copyleft` function. Now the candidate heap cells to deallocate inside the recursive call's body are $\pi.\mathsf{left} \mathbin{\dot{\setminus}} R$ (the heap cells for `t1` excluding those in the result of the recursive call). For each constraint $b \Rightarrow L$, the algorithm finds a boolean expression which guarantees that L and $\pi.\mathsf{left} \mathbin{\dot{\setminus}} R$ are disjoint if b is true; then, it takes the conjunction of all the found boolean expressions.

$\boxed{\mathsf{disjoint}_{\mathcal{B}}(L_1, L_2)}$ gives a condition that L_1 and L_2 are disjoint under $\mathcal{B}$. We apply the first available rule in the followings:

$\mathsf{disjoint}_{\mathcal{B}}(A, X) \stackrel{\Delta}{=} \text{true}$, and $\mathsf{disjoint}_{\mathcal{B}}(\emptyset, L) \stackrel{\Delta}{=} \text{true}$ (D1)

$\mathsf{disjoint}_{\mathcal{B}}(X_1, X_2) \stackrel{\Delta}{=} \text{true}$ (when $X_1 \neq X_2$) (D2)

$\mathsf{disjoint}_{\mathcal{B}}(\pi.\mathsf{root}, \pi.o) \stackrel{\Delta}{=} \text{true}$ (when $o = \mathsf{left}$ or right) (D3)

$\mathsf{disjoint}_{\mathcal{B}}(\pi.\mathsf{left}, \pi.\mathsf{right}) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(\mathcal{B}(\pi))$ (D4)

$\mathsf{disjoint}_{\mathcal{B}\cup\{R\mapsto L\}}(L_1 \dot{\setminus} R, L_2) \stackrel{\Delta}{=} \mathsf{disjoint}_{\mathcal{B}\cup\{R\mapsto\emptyset\}}(L_1, L_2)$ (D5)

$\mathsf{disjoint}_{\mathcal{B}}(R, L) \stackrel{\Delta}{=} \mathsf{disjoint}_{\mathcal{B}}(\mathcal{B}(R), L)$ (D6)

$\mathsf{disjoint}_{\mathcal{B}}(L_1 \dot{\sqcup} L_2, L_3) \stackrel{\Delta}{=} \mathsf{disjoint}_{\mathcal{B}}(L_1, L_3) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_2, L_3)$ (D7)

$\mathsf{disjoint}_{\mathcal{B}}(L_1 \dot{\oplus} L_2, L_3) \stackrel{\Delta}{=} \mathsf{disjoint}_{\mathcal{B}}(L_1, L_3) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_2, L_3)$ (D8)

$\mathsf{disjoint}_{\mathcal{B}}(\pi.o, L) \stackrel{\Delta}{=} \mathsf{disjoint}_{\mathcal{B}}(\mathcal{B}(\pi), L)$ (D9)

$\mathsf{disjoint}_{\mathcal{B}}(L_1, L_2) \stackrel{\Delta}{=} \text{false}$ (for other L_1 and L_2) (D10)

$\boxed{\mathsf{nosharing}_{\mathcal{B}}(L)}$ gives a condition that L is a *set* under $\mathcal{B}$:

$\mathsf{nosharing}_{\mathcal{B}}(A) \stackrel{\Delta}{=} \beta_{\text{ns}}$ (D11)
(where β_{ns} is the second dynamic flag of the enclosing function)

$\mathsf{nosharing}_{\mathcal{B}}(L) \stackrel{\Delta}{=} \text{true}$ (when $L = X$, $\pi.\mathsf{root}$, or $\emptyset$) (D12)

$\mathsf{nosharing}_{\mathcal{B}}(\pi.o) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(\mathcal{B}(\pi))$ (when $o = \mathsf{left}$ or right) (D13)

$\mathsf{nosharing}_{\mathcal{B}}(R) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(\mathcal{B}(R))$ (D14)

$\mathsf{nosharing}_{\mathcal{B}}(L_1 \dot{\sqcup} L_2) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(L_1) \wedge \mathsf{nosharing}_{\mathcal{B}}(L_2)$ (D15)

$\mathsf{nosharing}_{\mathcal{B}}(L_1 \dot{\oplus} L_2) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(L_1) \wedge \mathsf{nosharing}_{\mathcal{B}}(L_2) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_1, L_2)$ (D16)

$\mathsf{nosharing}_{\mathcal{B}}(L \dot{\setminus} R) \stackrel{\Delta}{=} \mathsf{nosharing}_{\mathcal{B}}(L)$ (D17)

Fig. 6. The Algorithm to Find a Condition for the Disjointness.

- For "$\neg\beta \Rightarrow A$," the algorithm in Figure 6 returns false for the condition that A and $\pi.\mathsf{left} \dot{\setminus} R$ are disjoint:

$$\begin{aligned}
\mathsf{disjoint}_{\mathcal{B}}(A, \pi.\mathsf{left} \dot{\setminus} R) &= \mathsf{disjoint}_{\mathcal{B}'}(A, \pi.\mathsf{left}) && \text{(excluding } R\text{)} && \text{(D5)}\\
&= \mathsf{disjoint}_{\mathcal{B}'}(A, A) && (\pi.\mathsf{root} \dot{\sqcup} (\pi.\mathsf{left} \dot{\oplus} \pi.\mathsf{right}) = A) && \text{(D9)}\\
&= \text{false} && (A = A) && \text{(D10)}
\end{aligned}$$

  where $\mathcal{B} = \{R \mapsto \pi.\mathsf{left} \dot{\sqcup} X_2, \pi \mapsto A\}$ and $\mathcal{B}' = \{R \mapsto \emptyset, \pi \mapsto A\}$. We take $\neg(\neg\beta) \vee \text{false}$, equivalently, β.
- For "true $\Rightarrow X_3 \dot{\sqcup} (R \oplus \pi.\mathsf{right})$," the algorithm in Figure 6 finds out that β_{ns} ensures the disjointness requirement:

$$\begin{aligned}
&\mathsf{disjoint}_{\mathcal{B}}(X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right}), \pi.\mathsf{left} \dot{\setminus} R)\\
&= \mathsf{disjoint}_{\mathcal{B}'}(X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\mathsf{right}), \pi.\mathsf{left}) && \text{(D5)}\\
&= \mathsf{disjoint}_{\mathcal{B}'}(X_3, \pi.\mathsf{left}) \wedge \mathsf{disjoint}_{\mathcal{B}'}(R, \pi.\mathsf{left}) \wedge \mathsf{disjoint}_{\mathcal{B}'}(\pi.\mathsf{right}, \pi.\mathsf{left}) && \text{(D7\&D8)}\\
&= \mathsf{disjoint}_{\mathcal{B}'}(X_3, A) \wedge \mathsf{disjoint}_{\mathcal{B}'}(\emptyset, \pi.\mathsf{left}) \wedge \mathsf{nosharing}_{\mathcal{B}'}(A) && \text{(D9\&D6\&D4)}\\
&= \text{true} \wedge \text{true} \wedge \beta_{\text{ns}} && \text{(D1\&D1\&D11)}
\end{aligned}$$

Thus the conjunction $\beta \wedge \beta_{\text{ns}}$ becomes the condition for the recursive call body to free a cell in its argument `t1`.

For the second boolean flag in the recursive call `(2)`, we find a boolean expression that ensures no sharing of a sub-part inside the left subtree `t1`. We use the memory-type $\pi.\mathsf{left}$ of `t1`, and find a boolean expression that guarantees no sharing inside the multiset $\pi.\mathsf{left}$; β_{ns} becomes such an expression: $\mathsf{nosharing}_{\mathcal{B}}(\pi.\mathsf{left}) = \mathsf{nosharing}_{\mathcal{B}}(A) = \beta_{\mathrm{ns}}$ (D13 & D11).

The algorithm inserts a `free` command right before "`Node (p,t2)`" at line `(3)`, which deallocates the root cell of the tree `t`. But the `free` command is safe only in certain circumstances: the cell should not already have been freed by the recursive call `(2)`, and the cell is neither freed nor used after the return of the current call. Our algorithm shows that we can meet all these requirements if the dynamic flag β is true; so, the algorithm picks β as a guard for the inserted `free` command. The process to find β is similar to the one for the first parameter of the call `(2)`. We first collect constraints about conditions for which heap cells we should not free:

- we should not free cells that can be freed before ($\beta \wedge \beta_{\mathrm{ns}} \Rightarrow \pi.\mathsf{left} \mathbin{\dot{\setminus}} R$),
- we should not free the input cells when β is false ($\neg\beta \Rightarrow A$), and
- we should not free cells that are included in the function's result (true $\Rightarrow X_3 \mathbin{\dot{\sqcup}} (R \mathbin{\dot{\oplus}} \pi.\mathsf{right})$).

These three constraints are generated by rules I-APP, I-FUN and I-FREE in Figure 5, respectively. From these constraints, we find a condition that cell $\pi.\mathsf{root}$ to free is disjoint with those cells we should not free. We use the same process as used for finding the first dynamic flag of the call `(2)`. The result is β.

Theorem 1 (Correctness). *For every well-typed closed expression e, when e is transformed to e' by the memory-usage analysis ($\emptyset \rhd e : \mathcal{B}, \mu, L$) and the* `free`*-insertion algorithm ($\mathcal{B}, \emptyset, \mathrm{false} \rhd e^{(\emptyset,\mathcal{B},\mu,L)} \Rightarrow e' : \mathcal{C}$), then expression e' is well-typed in the sound memory-type system in* [11].

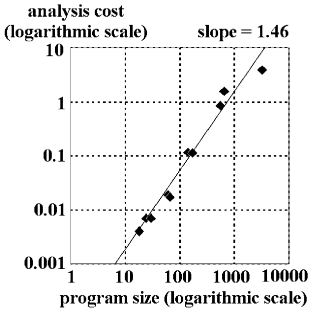
The complete proofs are in [11].

5 Experiments

We experimented the insertion algorithm with ML benchmark programs which use various data types such as lists, trees, and abstract syntax trees. We first pre-processed benchmark programs to monomorphic and closure-converted [13] programs, and then applied the algorithm to the pre-processed programs.

We extended the presented algorithm to analyze and transform programs with more features. (1) Our implementation supports more data constructors than just `Leaf` and `Node`. It analyzes heap cells with different constructors separately, and it inserts twice as many dynamic flags as the number of constructors for each parameter. (2) For functions with several parameters, we made the dynamic flag β also keep the alias information between function parameters so that if two parameters share some heap cells, both of their dynamic flags β are turned off. (3) For higher-order cases, we simply assumed the worst memory-types for the argument functions. For instance, we just assumed that an argument function, whose type is $\mathsf{tree} \to \mathsf{tree}$, has memory-type $\forall A.A \to \exists X.(L, L)$

program	lines	(1) total[a]	(2) reuse[a]	(2)/(1)	cost(s[b])
`sieve`[c]	18	161112	131040	81.3%	0.004
`quicksort`[d]	24	675925	617412	91.3%	0.007
`merge`[e]	30	120012	59997	50.0%	0.007
`mergesort`[d]	61	440433	390429	88.7%	0.019
`queens`[f]	66	118224	6168	5.2%	0.017
`mirage`[g]	141	208914	176214	84.4%	0.114
`life`[h]	169	84483	8961	10.6%	0.113
`kb`[h]	557	2747397	235596	8.6%	0.850
`k-eval`[i]	645	271591	161607	59.5%	1.564
`nucleic`[h]	3230	1616487	294067	18.2%	3.893



[a] words: the amount of total allocated heap cells and reused heap cells by our transformation
[b] seconds: our analysis and transformation is compiled by the Objective Caml 3.04 native compiler [12], and executed in Sun Sparc 400Mhz, Solaris 2.7
[c] prime number computation by the sieve of Eratosthenes (size = 10000)
[d] quick/merge sort of an integer list (size=10000)
[e] merging two ordered integer lists to an ordered list (size = 10000)
[f] eight queen problem
[g] an interpreter for a tiny non-deterministic programming language
[h] the benchmark programs from SML/NJ [19] benchmark suite (loop=50)
[i] a type-checker and interpreter for a tiny imperative programming language

Fig. 7. Experimental Results for Inserting Safe Deallocations.

where $L = (A \dot{\oplus} A) \dot{\sqcup} (X \dot{\oplus} X)$. (4) When we have multiple candidate cells for deallocation, we chose one whose guard is weaker than the others. For incomparable guards, we arbitrarily chose one.

The experimental results are shown in Figure 7. Our analysis and transformation achieves the memory reuse ratio (the fifth column) of 5.2% to 91.3%. For the two cases whose reuse ratio is low (`queens` and `kb`), we found that they have too much sharing. The `kb` program heavily uses a term-substitution function that can return a shared structure, where the number of shares depends on an argument value (e.g. a substitution item e/x has every x in the target term share e). Other than such cases, our experimental results are encouraging in terms of accuracy and cost. The graph in Figure 7 indicates that the analysis and transformation cost can be less than square in the program size in practice although the worst-case complexity is exponential.

6 Conclusion and Future Work

We have presented a static analysis and a source-level transformation that adds explicit memory-reuse commands into the program text, and we have shown that it effectively finds memory-reuse points.

We are currently implementing the analysis and transformation inside our nML compiler [15] to have it used in daily programming. The main issues in the implementation are to reduce the runtime overhead of the dynamic flags and to extend our method to handle polymorphism and mutable data structures. The runtime overhead of dynamic flags can be substantial because, for instance, if a function takes n parameters and each parameter's type has k data constructors, the function has to take $2 \times n \times k$ dynamic flags according to the current scheme. We are considering to reduce this overhead by doing a constant propagation for dynamic flags; omitting some unnecessary flags; associating a single flag with several data constructors of the same size; implementing flags by bit-vectors; and duplicating a function according to the different values of flags.

To extend our method for polymorphism, we need a sophisticated mechanism for dynamic flags. For instance, a polymorphic function of type $\forall\alpha.\, \alpha \to \alpha$ can take a value with two constructors or one with three constructors. So, this polymorphic input parameter does not fit in the current method because currently we insert twice as many dynamic flags as the number of constructors for each parameter. Our tentative solution is to assign only two flags to the input parameter of type α and to take conjunctions of flags in a call site: when a function is called with an input value with two constructors, instead of passing the four dynamic flags β, β_{ns}, β', and β'_{ns}, we pass $\beta \wedge \beta'$ and $\beta_{\mathrm{ns}} \wedge \beta'_{\mathrm{ns}}$. For mutable data structures, we plan to take a conservative approach similar to that of Gheorghioiu *et al.* [6]: heap cells possibly reachable from modifiable cells cannot be reused.

Acknowledgment We thank the anonymous referees for their helpful comments.

References

1. David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proceedings of the European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52, April 2002.
2. Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1995.
3. Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, 1998.
4. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 262–275, January 1999.
5. David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 70–80, June 2001.
6. Ovidiu Gheorghioiu, Alexandru Sălcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 273–284, January 2003.
7. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.
8. Williams L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

9. Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2001.
10. Naoki Kobayashi. Quasi-linear types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 29–42, 1999.
11. Oukseh Lee. A correctness proof on an algorithm to insert safe memory reuse commands. Tech. Memo. ROPAS-2003-19, Research On Program Analysis System, Korea Advanced Institute of Science and Technology, February 2003. `http://ropas.kaist.ac.kr/memo`.
12. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.04. Institut National de Recherche en Informatique et en Automatique, December 2001. `http://caml.inria.fr`.
13. Yosuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 271–283, January 1996.
14. Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. In *Proceedings of Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 241–258. Springer–Verlag, 1995.
15. nML programming language system, version 0.92a. Research On Program Analysis System, Korea Advanced Institute of Science and Technology, March 2002. `http://ropas.kaist.ac.kr/n`.
16. Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *The Proceedings of Computer Science and Logic*, pages 1–19, 2001.
17. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, July 2002.
18. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–382, March/April 2000.
19. The Standard ML of New Jersey, version 110.0.7. Bell Laboratories, Lucent Technologies, October 2000. `http://cm.bell-labs.com/cm/cs/what/smlnj`.
20. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.
21. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). IT University of Copenhagen, April 2002. `http://www.it-c.dk/research/mlkit`.
22. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.
23. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
24. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *International Conference on Functional Programming and Computer Architecture*, pages 25–28, June 1995.
25. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North Holland, April 1990.
26. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, September 2000.