Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 58 (2005) 141–178

Science of Computer Programming

www.elsevier.com/locate/scico

# Static insertion of safe and effective memory reuse commands into ML-like programs[☆]

Oukseh Lee[a,*], Hongseok Yang[b], Kwangkeun Yi[b]

[a] *Department of Computer Science & Engineering, Hanyang University, Ansan Gyeonggi 426-791, Republic of Korea*

[b] *School of Computer Science & Engineering, Seoul National University, Sillim-9-dong Gwanak-gu, Seoul 151-742, Republic of Korea*

Received 23 October 2003; received in revised form 31 August 2004; accepted 17 February 2005
Available online 24 May 2005

**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 system achieves a memory reuse ratio from 5.2% to 91.3% and reduces the memory peak from 0.0% to 71.9%. The small-ratio cases are for programs that have a number of data structures that are shared. For other cases, our experimental results are encouraging in terms of accuracy and cost. Major features of our analysis and transformation are: (1) polyvariant 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 cells as the granularity of explicit memory reuse. Our analysis and transformation system is fully automatic.
© 2005 Elsevier B.V. All rights reserved.

*Keywords:* Program analysis; Program transformation; Type system; Compile-time garbage collection

☆ Lee and Yi were supported by the Brain Korea 21 (2003–2004) of Korean Ministry of Education and Human Resource Development; Lee was supported by the research fund of Hanyang University (HY-2004-G); Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of the Korea Science and Engineering Foundation; and Yi was supported by Korea Research Foundation KRF-2003-041-D00528.

* Corresponding author.

*E-mail addresses:* oukseh@hanyang.ac.kr (O. Lee), hyang@ropas.snu.ac.kr (H. Yang), kwang@cse.snu.ac.kr (K. Yi).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.02.007

## 1. Overview

Our goal is to automatically insert explicit memory reuse commands into ML-like programs so that they do not blindly request memory when constructing data. We present a static analysis and a source-level transformation system that automatically adds explicit memory reuse commands into the program text. The explicit memory reuse is accomplished by inserting explicit memory-free commands right before data-construction expressions. Because the unit for deallocation and allocation is an individual cell, such deallocation 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`.

```
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 us 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 any longer, 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 the 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 variables `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 manage a reference counter for every heap cell.

Deductive systems like 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

[1] The drawback of this approach 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.

for our goal; they are not automatic. They need the programmer's help as regards memory invariants for loops or recursive functions.

The region-based memory managements [23,22,4,5,7] use a fixed partitioning strategy for recursive data structures, which is either implied by the programmer's region declarations or hardwired inside the region-inference engine [20,21]. Since every heap cell in a single region has the same lifetime, this "pre-determined" 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 in the relation's targets and the 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 a similar limitation: the deallocation is at the end of the function body.

## 1.2. Our solution

The features of our analysis and transformation are:

- Partitioning of heap cells is pivoted by two axes: by structures (e.g. heads and tails for lists, roots and subtrees for trees) and by set exclusions (e.g. cells A excluding B). This double-axis partitioning is expressive enough to isolate proper reusable cells from others.
- Sharing information among heap cells is maintained, in order to find the properties of disjointness between two partitions of heap cells. An analysis result consists of terms called "*multiset formulas*". A multiset formula symbolically manifests an abstract sharing relation between heap cells.
- The parameterized analysis result of a function is instantiated at each function call, in order to finalize the disjointness properties for the function's input and output. This polyvariant analysis is done without re-analyzing a function body multiple times.
- Dynamic flags are inserted into functions in order to condition their memory-free commands on their call sites. Dynamic flags are simple boolean expressions.

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 system is weak for programs that have too prevalent sharings among memory cells. Other than for those few cases, our experimental results are encouraging in terms of accuracy and cost: the reuse ratio ranges from 10.6% to 91.3% and the analysis cost ranges from about 400 to 4500 lines per second. The limitation is that we only consider ML-like immutable recursive data and a first-order monomorphic language without memory-free commands.

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 system works. Section 5 proves our analysis and transformation correct. Section 6 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 find few deallocation opportunities.

For a precise separation of two such 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) is not enough. We need to refine the partitions using the notion of exclusion. Consider a function that builds a tree from an input tree. Let us 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. However, we cannot free every node of the input 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 its input and its output are trees. The output tree's nodes along its leftmost 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 a 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 behaviors. By our `free` commands insertion, the above `copyleft` function is transformed to

```
fun copyleft [$\beta$, $\beta_{ns}$] t =
  case t of
    Leaf         => Leaf
  | Node (t1,t2) => let p = copyleft [$\beta \wedge \beta_{ns}$, $\beta_{ns}$] t1
                    in  (free t when $\beta$; Node (p,t2))
```

Flag $\beta$ 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` $\beta$". By the recursive calls, all the nodes along the leftmost 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

SYNTAX

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

OPERATIONAL SEMANTICS

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

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

Fig. 1. The syntax and the semantics.

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 $\beta$ is $\beta \wedge \beta_{\text{ns}}$, where flag $\beta_{\text{ns}}$ is true when there is no sharing inside the input tree.

## 2. Language

Fig. 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), deconstructions (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: `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 `free` $v$ `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 $\mathsf{tree}$ or $\mathsf{tree}\to\mathsf{tree}$. In our implementation, we handle higher-order functions, and arbitrary algebraic data types, not just binary trees. We explain more on this in Section 6.

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 system use what we call *memory-types* to estimate the heap objects for expression 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 them “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 via the following grammar:

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

Symbols $A$, $R$, $X$ and $\pi$ are just names for multisets of locations. $A$ symbolically denotes the heap cells in the input tree of a function, $X$ the newly allocated heap cells, $R$ the heap cells in the result tree of a function, and $\pi$ the heap objects whose roots and left/right subtrees are respectively $\pi.\text{root}$, $\pi.\text{left}$, and $\pi.\text{right}$. $\emptyset$ means the empty multiset, and the 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 including two occurrences of a location just if $L$ or $L'$ already means including two occurrences of the same location. The term $L \mathbin{\dot\setminus} L'$ means multiset $L$ excluding the locations included in $L'$.

Fig. 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 $\eta$ of name $X$, $A$, and $\pi$ in Fig. 2. The pre-order for $L$ is

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

SEMANTICS OF MULTISET FORMULAS

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

For all $\eta$ mapping $X$, $A$, $R$, $\pi$.root, $\pi$.left, and $\pi$.right to MultiSets,

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

where

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

REQUIREMENTS ON GOOD ENVIRONMENTS

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

SEMANTICS OF MEMORY-TYPES FOR TREES

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

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

Fig. 2. The semantics of multiset formulas and memory-types for trees.

### 3.2. Memory-types

Memory-types are given in terms of the multiset formulas. We define memory-types $\mu_\tau$ for value-type $\tau$ using multiset formulas:

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

A memory-type $\mu_{\text{tree}}$ 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 $L$, $\mu_1$, and $\mu_2$ 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 Fig. 2.

During our analysis, we switch between a structured memory-type and a collapsed memory-type. We can collapse a structured one via the collapse function:

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

Note that when combining $L$ and $\mathsf{collapse}(\mu_1) \mathbin{\dot\oplus} \mathsf{collapse}(\mu_2)$, we use $\dot\sqcup$ instead of $\dot\oplus$: this 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 the splitting name $\pi$:

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

The second component of the result of reconstruct is a resulting structured memory-type and the first one is a record that $L$ is a collection of $\pi$.root, $\pi$.left, and $\pi$.right. The pre-order $\sqsubseteq_{\mathsf{tree}}$ for memory-types for trees is

$$\begin{aligned}
L \sqsubseteq_{\mathsf{tree}} L' &\text{ iff } L \sqsubseteq L'\\
\langle L, \mu_1, \mu_2\rangle \sqsubseteq_{\mathsf{tree}} \langle L', \mu_1', \mu_2'\rangle &\text{ iff } L \sqsubseteq L', \mu_1 \sqsubseteq_{\mathsf{tree}} \mu_1', \text{ and } \mu_2 \sqsubseteq_{\mathsf{tree}} \mu_2'\\
\langle L, \mu_1, \mu_2\rangle \sqsubseteq_{\mathsf{tree}} L' &\text{ iff } \mathsf{collapse}(\langle L, \mu_1, \mu_2\rangle) \sqsubseteq_{\mathsf{tree}} L'
\end{aligned}$$

Note that this order is sound with respect to the semantics: if $\mu_1 \sqsubseteq_{\mathsf{tree}} \mu_2$, then $\forall\eta.\mathsf{goodEnv}(\eta) \implies [\![\mu_1]\!]_{\mathsf{tree}}\eta \subseteq [\![\mu_2]\!]_{\mathsf{tree}}\eta$. The join of two memory-types is done by an operator $\uplus$ that returns an upper bound[2] of two memory-types. The operator $\uplus$ is defined using the function collapse:

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

For a function type tree $\to$ 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.

[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.

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 that 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:

```
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 into the program.

### 4.1. Step one: The memory usage analysis

Our memory usage analysis (shown in Fig. 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 $\pi$: $A$ 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 tree returned from the recursive call "`copyleft t1`" at line `(2)`; the splitting name $\pi$ 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)` of the example: The `Leaf`-branch is executed only when `t` is `Leaf` whose memory-type is $\emptyset$. So, we assume that `t`'s memory-type is $\emptyset$ when analyzing the `Leaf`-branch (U-CASE).

  The memory-type for `Leaf` is $\emptyset$, which says that the result tree of `Leaf`-branch is empty (U-LEAF and U-VALUE).
- Line `(2)` of the example: 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:

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

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

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

$$\frac{\begin{array}{l}\Delta \rhd e_1 : \mathcal{B}_1, \mu_1, L_1 \\ \Delta \cup \{x \mapsto \mu_1\} \rhd e_2 : \mathcal{B}_2, \mu_2, L_2\end{array}}{\Delta \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathcal{B}_1 \cup \mathcal{B}_2, \mu_2, L_1 \dot{\sqcup} L_2}\ (\text{U-LET})$$

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

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

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

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

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

Fig. 3. Step one: the memory usage analysis.

the root cell named $\pi$.root, the left subtree named $\pi$.left, and the right subtree named $\pi$.right, and record that their collection is $A$: $\pi.\text{root} \dot{\sqcup} (\pi.\text{left} \oplus \pi.\text{right}) = A$. Then `t1` and `t2` have $\pi$.left and $\pi$.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 $\pi$.left of the argument `t1`, and $X$ by the name $X_2$ for the newly allocated cells at line (2). The instantiated memory-type $\pi.\text{left} \to (\pi.\text{left} \dot{\sqcup} X_2,\ \pi.\text{left})$ 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 $\pi.\text{left} \mathbin{\dot\sqcup} X_2$, and uses the cells in $\pi.\text{left}$. However, we use name $R$ for the result of the function call, and record that $R$ is included in $\pi.\text{left} \mathbin{\dot\sqcup} X_2$ (U-APP).

- Line `(3)` of the example: While analyzing line `(2)`, we have computed the memory-types of `p` and `t2`, that is, $R$ and $\pi.\text{right}$, respectively. Therefore, "`Node (p,t2)`" has the memory-type $\langle X_3,\ R,\ \pi.\text{right}\rangle$ where $X_3$ is a name for the newly allocated root cell at line `(3)`, $R$ for the left subtree, and $\pi.\text{right}$ for the right subtree (U-NODE).

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

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

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

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

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 $A \mathbin{\dot\sqcup} X$. By rules (w4&w3) in Fig. 4, $\pi.\text{root} \mathbin{\dot\sqcup} \pi.\text{left}$ 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 $A \mathbin{\dot\sqcup} A$. 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

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

*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.

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

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

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

Fig. 4. The widening process.

tree cannot 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 Fig. 5) inserts `free` commands, and adds boolean parameters $\beta$ and $\beta_{\text{ns}}$ (called *dynamic flags*) to each function. The dynamic flag $\beta$ says that a cell in the argument tree can be safely deallocated, and $\beta_{\text{ns}}$ that no sub-parts of the argument tree are shared. We have designed the transformation algorithm on the basis of 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.

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

*Preservation Constraints* $\quad \mathcal{E} \subseteq \{b \hookrightarrow L \mid b \text{ is a boolean expression}\}$

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

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

$\boxed{\mathcal{B}, \mathcal{E}_1, b \rhd e_1^{(\Delta,\mathcal{B}',\mu,L)} \Rightarrow e_2 : \mathcal{E}_2}$ 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 $\mathcal{E}_1$ that prohibit certain cells from being freed: $b$ says that the result of $e_1$ should not be freed, and each $b' \hookrightarrow L'$ in $\mathcal{E}_1$ says that $L'$ should not be freed when $b'$ holds. The algorithm returns a `free`-inserted $e_2$ and $\mathcal{E}_2$ whose $b' \hookrightarrow L'$ expresses that $L'$ is freed in $e_2$ when $b'$ holds.

$$\frac{\rhd\, v \Rightarrow v'}{\mathcal{B}, \mathcal{C}, b \rhd v \Rightarrow v' : \emptyset}\ (\text{I-VALUE})$$

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

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

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

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

$$\frac{\rhd\, v \Rightarrow v' \quad L \triangleq \mathsf{collapse}(\mu) \quad b \triangleq \mathsf{freeCond}_{\mathcal{B},\mathcal{E}}(L \dot{\setminus} R) \quad b_{\text{ns}} \triangleq \mathsf{noSharing}_{\mathcal{B}}(L)}{\mathcal{B}, \mathcal{E}, b' \ \rhd\ (x\ (v^{(\Delta,\mu)}))^{(\cdot,\cdot,R,\cdot)} \Rightarrow x\ [b, b_{\text{ns}}]\ v' : \{b \hookrightarrow L \dot{\setminus} R\}}\ (\text{I-APP})$$

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

$$\mathsf{freeCond}_{\mathcal{B},\mathcal{E}}(L) \triangleq \bigwedge \{\neg b \vee \mathsf{disjoint}_{\mathcal{B}}(L, L') \mid (b \hookrightarrow L') \in \mathcal{E}\}$$

Fig. 5. Step two: the algorithm for inserting `free` commands.

Our algorithm transforms the `copyleft` function as follows:

```
fun copyleft [$\beta$, $\beta_{ns}$] t =
  case t of Leaf         => Leaf                                      (1)
          | Node (t1,t2) => let p = copyleft [$\beta \wedge \beta_{ns}$, $\beta_{ns}$] t1     (2)
                            in  (free t when $\beta$; 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 $\beta \wedge \beta_{ns}$ and $\beta_{ns}$ in the recursive call `(2)` (rule I-APP). To find the first parameter, we collect constraints about conditions for which heap cells we should not free ($\mathcal{E}$ in I-APP). Then, the candidate heap cells to deallocate must be disjoint from the cells to preserve. We derive such a disjointness condition, expressed by a simple boolean expression ($\mathsf{freeCond}_{\mathcal{B},\mathcal{E}}(L \dot{\setminus} R)$ in I-APP). A preservation constraint has the conditional form $b \hookrightarrow L$: 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 "$\neg\beta \hookrightarrow A$" and "true $\hookrightarrow X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\text{right})$" from the algorithm in Fig. 5 (rules I-FUN and I-LET). The first constraint means that we should not free the cells in the argument tree `t` if $\beta$ 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 $\pi.\text{left} \dot{\setminus} R$ (the heap cells for `t1` excluding those in the result of the recursive call). For each constraint $b \hookrightarrow L$, the algorithm finds a boolean expression which guarantees that $L$ and $\pi.\text{left} \dot{\setminus} R$ are disjoint if $b$ is true; then, it takes the conjunction of all the boolean expressions found.

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

$$\begin{array}{lllr}
\multicolumn{3}{l}{\mathsf{disjoint}_{\mathcal{B}}(A, \pi.\text{left} \dot{\setminus} R)} & \\
= & \mathsf{disjoint}_{\mathcal{B}'}(A, \pi.\text{left}) & (\text{excluding } R) & (\text{D5}) \\
= & \mathsf{disjoint}_{\mathcal{B}'}(A, A) & (\pi.\text{root} \dot{\sqcup} (\pi.\text{left} \dot{\oplus} \pi.\text{right}) = A) & (\text{D9}) \\
= & \text{false} & (A = A) & (\text{D10})
\end{array}$$

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

$$\begin{array}{llr}
\multicolumn{2}{l}{\mathsf{disjoint}_{\mathcal{B}}(X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\text{right}), \pi.\text{left} \dot{\setminus} R)} & \\
= & \mathsf{disjoint}_{\mathcal{B}'}(X_3 \dot{\sqcup} (R \dot{\oplus} \pi.\text{right}), \pi.\text{left}) & (\text{D5}) \\
= & \mathsf{disjoint}_{\mathcal{B}'}(X_3, \pi.\text{left}) \wedge \mathsf{disjoint}_{\mathcal{B}'}(R, \pi.\text{left}) \wedge \mathsf{disjoint}_{\mathcal{B}'}(\pi.\text{right}, \pi.\text{left}) & \\
 & & (\text{D7\&D8}) \\
= & \mathsf{disjoint}_{\mathcal{B}'}(X_3, A) \wedge \mathsf{disjoint}_{\mathcal{B}'}(\emptyset, \pi.\text{left}) \wedge \mathsf{noSharing}_{\mathcal{B}'}(A) & (\text{D9\&D6\&D4}) \\
= & \text{true} \wedge \text{true} \wedge \beta_{ns} & (\text{D1\&D1\&D11})
\end{array}$$

Thus the conjunction $\beta \wedge \beta_{ns}$ 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` ($\mathsf{noSharing}_{\mathcal{B}}(L)$ in I-APP). We use the memory-type $\pi.\text{left}$ of `t1`, and find a boolean expression that guarantees no

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

$$\mathsf{disjoint}_{\mathcal{B}}(A, X) \triangleq \text{true, and } \mathsf{disjoint}_{\mathcal{B}}(\emptyset, L) \triangleq \text{true} \qquad (\text{D}1)$$

$$\mathsf{disjoint}_{\mathcal{B}}(X_1, X_2) \triangleq \text{true} \qquad (\text{when } X_1 \neq X_2) \qquad (\text{D}2)$$

$$\mathsf{disjoint}_{\mathcal{B}}(\pi.\text{root}, \pi.o) \triangleq \text{true} \qquad (\text{when } o = \text{left or right}) \qquad (\text{D}3)$$

$$\mathsf{disjoint}_{\mathcal{B}}(\pi.\text{left}, \pi.\text{right}) \triangleq \mathsf{noSharing}_{\mathcal{B}}(\mathcal{B}(\pi)) \qquad (\text{D}4)$$

$$\mathsf{disjoint}_{\mathcal{B}\cup\{R\mapsto L\}}(L_1 \dot{\setminus} R, L_2) \triangleq \mathsf{disjoint}_{\mathcal{B}\cup\{R\mapsto\emptyset\}}(L_1, L_2) \qquad (\text{D}5)$$

$$\mathsf{disjoint}_{\mathcal{B}}(R, L) \triangleq \mathsf{disjoint}_{\mathcal{B}}(\mathcal{B}(R), L) \qquad (\text{D}6)$$

$$\mathsf{disjoint}_{\mathcal{B}}(L_1 \dot{\sqcup} L_2, L_3) \triangleq \mathsf{disjoint}_{\mathcal{B}}(L_1, L_3) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_2, L_3) \qquad (\text{D}7)$$

$$\mathsf{disjoint}_{\mathcal{B}}(L_1 \dot{\oplus} L_2, L_3) \triangleq \mathsf{disjoint}_{\mathcal{B}}(L_1, L_3) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_2, L_3) \qquad (\text{D}8)$$

$$\mathsf{disjoint}_{\mathcal{B}}(\pi.o, L) \triangleq \mathsf{disjoint}_{\mathcal{B}}(\mathcal{B}(\pi), L) \qquad (\text{D}9)$$

$$\mathsf{disjoint}_{\mathcal{B}}(L_1, L_2) \triangleq \text{false} \qquad (\text{for other } L_1 \text{ and } L_2) \qquad (\text{D}10)$$

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

$$\mathsf{noSharing}_{\mathcal{B}}(A) \triangleq \beta_{\text{ns}} \qquad (\text{D}11)$$

(where $\beta_{\text{ns}}$ is the second dynamic flag of the enclosing function)

$$\mathsf{noSharing}_{\mathcal{B}}(L) \triangleq \text{true} \qquad (\text{when } L = X, \pi.\text{root, or } \emptyset) \qquad (\text{D}12)$$

$$\mathsf{noSharing}_{\mathcal{B}}(\pi.o) \triangleq \mathsf{noSharing}_{\mathcal{B}}(\mathcal{B}(\pi)) \qquad (\text{when } o = \text{left or right}) \qquad (\text{D}13)$$

$$\mathsf{noSharing}_{\mathcal{B}}(R) \triangleq \mathsf{noSharing}_{\mathcal{B}}(\mathcal{B}(R)) \qquad (\text{D}14)$$

$$\mathsf{noSharing}_{\mathcal{B}}(L_1 \dot{\sqcup} L_2) \triangleq \mathsf{noSharing}_{\mathcal{B}}(L_1) \wedge \mathsf{noSharing}_{\mathcal{B}}(L_2) \qquad (\text{D}15)$$

$$\mathsf{noSharing}_{\mathcal{B}}(L_1 \dot{\oplus} L_2) \triangleq \mathsf{noSharing}_{\mathcal{B}}(L_1) \wedge \mathsf{noSharing}_{\mathcal{B}}(L_2) \wedge \mathsf{disjoint}_{\mathcal{B}}(L_1, L_2) \qquad (\text{D}16)$$

$$\mathsf{noSharing}_{\mathcal{B}}(L \dot{\setminus} R) \triangleq \mathsf{noSharing}_{\mathcal{B}}(L) \qquad (\text{D}17)$$

Fig. 6. The algorithm for finding a condition for the disjointness.

sharing inside the multiset $\pi$.left; $\beta_{\text{ns}}$ becomes such an expression: $\mathsf{noSharing}_{\mathcal{B}}(\pi.\text{left}) = \mathsf{noSharing}_{\mathcal{B}}(A) = \beta_{\text{ns}}$ (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` (I-FREE). 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 $\beta$ is true; so, the algorithm picks $\beta$ as a guard for the inserted `free` command. The process for finding $\beta$ 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 ($\beta \wedge \beta_{\text{ns}} \hookrightarrow \pi.\text{left}\dot{\setminus}R$), SEMANTICS OF SAFETY CONSTRAINTS: $\eta \models \mathcal{C}$

$$\begin{array}{ll}
\eta \models \text{SET}(L) & \text{iff } [\![L]\!]\eta \sqsubseteq \lambda l.1 \\
\eta \models L_1 \# L_2 & \text{iff } ([\![L_1]\!]\eta) \sqcap ([\![L_2]\!]\eta) = \bot \\
\eta \models L_1 \sqsubseteq_{\text{set}} L_2 & \text{iff } ([\![L_1]\!]\eta) \sqcap \lambda l.1 \sqsubseteq [\![L_2]\!]\eta \\
\eta \models L_1 \sqsubseteq L_2 & \text{iff } [\![L_1]\!]\eta \sqsubseteq [\![L_2]\!]\eta \\
\eta \models \mathcal{E}_1 \sqsubseteq \mathcal{E}_2 & \text{iff } \eta \models L_1 \sqsubseteq_{\text{set}} L_2 \text{ where } L_i = \dot\sqcup\{L \mid (b \hookrightarrow L) \in \mathcal{E}_i,\ b \not\Leftrightarrow \text{false}\} \\
\\
\eta \models \text{true} & \text{always} \\
\eta \models b \Rightarrow \mathcal{C} & \text{iff } (b \Leftrightarrow \text{false}) \vee (\eta \models \mathcal{C}) \\
\eta \models \mathcal{C}_1 \wedge \mathcal{C}_2 & \text{iff } (\eta \models \mathcal{C}_1) \wedge (\eta \models \mathcal{C}_2)
\end{array}$$

Fig. 7. The semantics of the safety constraints.

- we should not free the input cells when $\beta$ is false ($\neg\beta \hookrightarrow A$), and
- we should not free cells that are included in the function's result (true $\hookrightarrow X_3 \dot\sqcup (R \dot\oplus \pi.\text{right})$).

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

## 5. Algorithm correctness

The correctness of our analysis and transformation is proved via a type system for safe memory deallocations. In Section 5.1, we introduce a memory-type system, and in Section 5.2, we prove that our memory-type system is sound: every well-typed program in the system does not access any deallocated heap cells. Then in Section 5.3, we prove 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.

### 5.1. The memory-type system

We use a safety constraint in our type system for the memory safety of programs. For instance, consider that a function takes a tree as its input, deallocates all of its right subtree, and then accesses its left subtree. For such a function, our type system deduces that its input tree must have no shared sub-parts between its left and right subtrees. This judgment is expressed by the following safety constraint:

$$\begin{array}{lcl}
p & ::= & \text{SET}(L) \mid L\#L \mid L \sqsubseteq_{\text{set}} L \mid L \sqsubseteq L \mid \mathcal{E} \sqsubseteq \mathcal{E} \\
\mathcal{C} & ::= & p \mid b \Rightarrow \mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \text{true} \mid \text{false}
\end{array}$$

The exact semantic definition of $\mathcal{C}$ is in Fig. 7, and the definition of the multiset formula $L$ is in Section 3.1. Predicate SET$(L)$ means that a multiset formula $L$ is indeed a set

SYNTACTIC SUGARS

$$
\begin{aligned}
\pi \sqsubseteq L &\triangleq \pi.\text{root} \mathbin{\dot\sqcup} (\pi.\text{left} \oplus \pi.\text{right}) \sqsubseteq L\\
\textsc{precise}(\langle L, \mu_1, \mu_2\rangle) &\triangleq \textsc{set}(L) \wedge (L \# \mathsf{collapse}(\mu_1)) \wedge (L \# \mathsf{collapse}(\mu_2))\\
\textsc{precise}(L) &\triangleq \text{true}\\
\mathcal{E}\#L = L\#\mathcal{E} &\triangleq \bigwedge \left\{ b \Rightarrow L\#L' \;\middle|\; (b \hookrightarrow L') \in \mathcal{E} \right\}\\
\mathcal{E}_1\#\mathcal{E}_2 &\triangleq \bigwedge \left\{ b_1 \wedge b_2 \Rightarrow L_1\#L_2 \;\middle|\; \begin{array}{l}(b_1 \hookrightarrow L_1) \in \mathcal{E}_1,\\ (b_2 \hookrightarrow L_2) \in \mathcal{E}_2\end{array} \right\}\\
\mathcal{B} &\triangleq \bigwedge \{ V \sqsubseteq \mathcal{B}(V) \mid V \in \text{dom}(\mathcal{B}) \}\\
L \sqsubseteq_{\mathsf{tree}} L' &\triangleq L \sqsubseteq L'\\
\langle L_1, \mu_1, \mu_2\rangle \sqsubseteq_{\mathsf{tree}} \langle L_2, \mu_1', \mu_2'\rangle &\triangleq (L_1 \sqsubseteq L_2) \wedge (\mu_1 \sqsubseteq_{\mathsf{tree}} \mu_2) \wedge (\mu_1' \sqsubseteq_{\mathsf{tree}} \mu_2')\\
\langle L, \mu_1, \mu_2\rangle \sqsubseteq_{\mathsf{tree}} L' &\triangleq \mathsf{collapse}(\langle L, \mu_1, \mu_2\rangle) \sqsubseteq L'\\
L' \sqsubseteq_{\mathsf{tree}} \langle L, \mu_1, \mu_2\rangle &\triangleq \text{false}\\
\mu \sqsubseteq_{\mathsf{tree}\to\mathsf{tree}} \mu' &\triangleq \begin{cases}\text{true}, & \text{if they are } \alpha\text{-equivalent},\\ \text{false}, & \text{otherwise.}\end{cases}\\
\mu \sqsubseteq \mu' &\triangleq \begin{cases}\mu \sqsubseteq_{\mathsf{tree}} \mu', & \text{for memory-types for trees},\\ \mu \sqsubseteq_{\mathsf{tree}\to\mathsf{tree}} \mu', & \text{for memory-types for functions.}\end{cases}
\end{aligned}
$$

Fig. 8. The syntactic sugars of the safety constraints.

(i.e., a tree in $L$ has no shared sub-part), $L_1\#L_2$ means that $L_1$ and $L_2$ are disjoint, $L_1 \sqsubseteq L_2$ means that multiset $L_2$ includes multiset $L_1$, $L_1 \sqsubseteq_{\mathsf{set}} L_2$ means that if we interpret them as sets, $L_1$ is a subset of $L_2$, i.e., every location in $L_1$ is also in $L_2$, and $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2$ means that $\mathcal{E}_2$ says more deallocations than $\mathcal{E}_1$ does. Constraint $\mathcal{C}$ holds if and only if for any substitution $\mathcal{S}$ for the boolean variables,

$$\forall \eta.\mathsf{goodEnv}(\eta) \Longrightarrow (\eta \models \mathcal{S}\mathcal{C}).$$

Constraint $\mathcal{C}_1$ is stronger than constraint $\mathcal{C}_2$ ($\mathcal{C}_1 \Rightarrow \mathcal{C}_2$) if and only if, for any substitution $\mathcal{S}$ for the boolean variables,

$$\forall \eta.\mathsf{goodEnv}(\eta) \wedge (\eta \models \mathcal{S}\mathcal{C}_1) \Longrightarrow (\eta \models \mathcal{S}\mathcal{C}_2).$$

In Fig. 8, we define some notation and make it clear that the bound $\mathcal{B}$ (a map from names to a multiset formula, Fig. 3) and the pre-order relation $\sqsubseteq_{\mathsf{tree}}$ (in Section 3.2) of memory-types for trees are expressed in our constraints.

By using a safety constraint, we define the memory-types for functions as

$$\mu_{\mathsf{tree}\to\mathsf{tree}} ::= \lambda\beta.\lambda\beta_{\text{ns}}.\lambda A.\exists\mathcal{V}.\,(\mathcal{B}, \mu_{\mathsf{tree}}, L, \mathcal{E})\,\&\,\mathcal{C}.$$

A function takes two boolean parameters $\beta$ and $\beta_{\text{ns}}$ and one $\mathsf{tree}$-typed value named $A$. When constraint $\mathcal{C}$ is satisfied, the function can access only the heap cells in $L$, can

SUBSTITUTION

$$\begin{aligned}\mathcal{S} \subseteq\ & \{L/V \mid V \text{ is } A, X, R, \pi.\text{root}, \pi.\text{left, or } \pi.\text{right},\ L \text{ is a multiset formula}\} \cup \\ & \{b/\beta \mid \beta \text{ is a boolean variable, } b \text{ is a boolean expression}\}\end{aligned}$$

where

$$\begin{aligned}\text{supp}(\mathcal{S}) = & \{V \mid (L/V) \in \mathcal{S},\ V \text{ is } A, X, \text{ or } R\} \cup \\ & \{\pi \mid (L/\pi.\text{root}), (L/\pi.\text{left}), \text{ or } (L/\pi.\text{right}) \in \mathcal{S}\} \cup \\ & \{\beta \mid (b/\beta) \in \mathcal{S}\}\end{aligned}$$

APPLYING A SUBSTITUTION

$$\begin{aligned}
\mathcal{S}\mu_{\mathsf{tree}} &= \begin{cases}\mathcal{S}L, & \text{if } \mu_{\mathsf{tree}} = L \\ \langle \mathcal{S}L, \mathcal{S}\mu_1, \mathcal{S}\mu_2\rangle, & \text{if } \mu_{\mathsf{tree}} = \langle L, \mu_1, \mu_2\rangle\end{cases} \\
\mathcal{S}\mu_{\mathsf{tree}\to\mathsf{tree}} &= \mu_{\mathsf{tree}\to\mathsf{tree}} \\
\mathcal{S}\Delta &= \{id \mapsto \mathcal{S}\mu \mid (id \mapsto \mu) \in \Delta\} \\
\mathcal{S}\mathcal{B} &= \begin{cases}\{V \mapsto \mathcal{S}L \mid (V \mapsto L) \in \mathcal{B}\}, & \text{if supp}(\mathcal{S}) \cap \text{dom}(\mathcal{B}) = \emptyset \\ \mathcal{S}(\wedge_{V \in \text{dom}(\mathcal{B})} V \sqsubseteq \mathcal{B}(V)), & \text{otherwise}\end{cases} \\
\mathcal{S}\mathcal{E} &= \{\mathcal{S}b \hookrightarrow \mathcal{S}L \mid (b \hookrightarrow L) \in \mathcal{E}\} \\
\mathcal{S}\mathcal{C} &= \begin{cases}\text{SET}(\mathcal{S}L), & \text{if } \mathcal{C} = \text{SET}(L) \\ (\mathcal{S}L_1)\ op\ (\mathcal{S}L_2), & \text{if } \mathcal{C} = L_1\ op\ L_2 \text{ where } op = \#,\ \sqsubseteq_{\mathsf{set}},\ \text{or } \sqsubseteq \\ \mathcal{S}b \Rightarrow \mathcal{S}\mathcal{C}', & \text{if } \mathcal{C} = b \Rightarrow \mathcal{C}' \\ (\mathcal{S}\mathcal{C}_1) \wedge (\mathcal{S}\mathcal{C}_2), & \text{if } \mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2 \\ \mathcal{C}, & \text{if } \mathcal{C} = \text{true or false}\end{cases}
\end{aligned}$$

Fig. 9. Substitution.

deallocate only those in $\mathcal{E}$, and returns a result that has memory-type $\mu_{\mathsf{tree}}$. Set $\mathcal{V}$ is the set of new names that appear in the type, and $\mathcal{B}$ imposes conditions on those names. Since we assume that every function is closed, we consider only closed memory-types: every name or boolean variable is either $\beta$, $\beta_{\text{ns}}$, $A$, or the names in $\mathcal{V}$.

We have a mapping from the memory-types in the algorithm to those in the memory-type system:

$$\begin{aligned}
\mathcal{T}(\mu_{\mathsf{tree}}) &= \mu_{\mathsf{tree}} \\
\mathcal{T}(\forall A.A \to \exists X.(L_1, L_2)) &= \lambda\beta.\lambda\beta_{\text{ns}}.\lambda A.\exists\{X, R\}. \\
&\qquad \big(\{R \mapsto L_1\}, R, L_2, \{\beta \hookrightarrow A \dot{\setminus} R,\ \text{true} \hookrightarrow X \dot{\setminus} R\}\big) \\
&\qquad \&(\beta_{\text{ns}} \Rightarrow \text{SET}(A)) \\
\mathcal{T}(\Delta) &= \{x \mapsto \mathcal{T}(\Delta(x)) \mid x \in \text{dom}(\Delta)\}
\end{aligned}$$

Our plan of program transformation is manifest in this translation: (1) we do not deallocate the heap cells in the result ($A \dot{\setminus} R$ and $X \dot{\setminus} R$); (2) only when $\beta$ is true we deallocate the input tree ($\beta \hookrightarrow A \dot{\setminus} R$); and (3) $\beta_{\text{ns}}$ should indicate that the input has no shared sub-part ($\beta_{\text{ns}} \Rightarrow \text{SET}(A)$).

The memory-type system is defined in Figs. 11–13. In the definition, we use substitutions (Fig. 9) and the function “free” in Fig. 10 which gives a set of free names

FREE NAMES

$$\mathrm{free}(L) = \begin{cases} \{L\}, & \text{if } L = A,\ X,\ \text{or } R \\ \{\pi\}, & \text{if } L = \pi.\mathrm{root},\ \pi.\mathrm{left},\ \text{or } \pi.\mathrm{right} \\ \mathrm{free}(L_1) \cup \mathrm{free}(L_2), & \text{if } L = L_1 \dot{\cup} L_2,\ L_1 \dot{\oplus} L_2,\ \text{or } L_1 \dot{\setminus} L_2 \\ \emptyset, & \text{if } L = \emptyset \end{cases}$$

$$\mathrm{free}(\mu_{\mathsf{tree}}) = \begin{cases} \mathrm{free}(L), & \text{if } \mu_{\mathsf{tree}} = L \\ \mathrm{free}(L) \cup \mathrm{free}(\mu_1) \cup \mathrm{free}(\mu_2), & \text{if } \mu_{\mathsf{tree}} = \langle L, \mu_1, \mu_2 \rangle \end{cases}$$

$$\mathrm{free}(\mu_{\mathsf{tree}\to\mathsf{tree}}) = \emptyset$$

$$\mathrm{free}(\Delta) = \bigcup \{\mathrm{free}(\mu) \mid (id \mapsto \mu) \in \Delta\}$$

$$\mathrm{free}(\mathcal{B}) = \bigcup \{\mathrm{free}(L) \cup \{V\} \mid (V \mapsto L) \in \mathcal{B}\}$$

$$\mathrm{free}(\mathcal{E}) = \bigcup \{\mathrm{free}(L) \mid (b \hookrightarrow L) \in \mathcal{E}\}$$

$$\mathrm{free}(\mathcal{C}) = \begin{cases} \mathrm{free}(L), & \text{if } \mathcal{C} = \textsc{set}(L) \\ \mathrm{free}(L_1) \cup \mathrm{free}(L_2), & \text{if } \mathcal{C} = L_1\ op\ L_2 \text{ for } op = \#,\ \sqsubseteq_{\mathsf{set}},\ \text{or } \sqsubseteq \\ \mathrm{free}(\mathcal{C}'), & \text{if } \mathcal{C} = b \Rightarrow \mathcal{C}' \\ \mathrm{free}(\mathcal{C}_1) \cup \mathrm{free}(\mathcal{C}_2), & \text{if } \mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2 \\ \emptyset, & \text{if } \mathcal{C} = \text{true or false} \end{cases}$$

$$\mathrm{free}(A_1, \ldots, A_n) = \bigcup_i \mathrm{free}(A_i)$$

Fig. 10. Free names.

in the arguments. Typing judgment "$\Delta \vdash v : \mu \,\&\, \mathcal{C}$" for a value $v$ (in Fig. 11) means that for a given memory-type environment $\Delta$, value $v$ has memory-type $\mu$ under constraint $\mathcal{C}$. A `Leaf`-value has a memory-type equal to or greater than $\emptyset$ (LEAF). An identifier *id* (a variable or a location) has a memory-type equal to or greater than $\Delta(id)$ (ID). The memory-type of a function value follows the result of its function body (FUN).

Typing judgment "$\Delta \vdash e : \exists \mathcal{V}.\, (\mathcal{B}, \mu, L, \mathcal{E}) \,\&\, \mathcal{C}$" for an expression $e$ (in Fig. 11) means that for a given memory-type environment $\Delta$, if constraint $\mathcal{C}$ is satisfied and the heap cells in $L$ and $\mathcal{E}$ are available, program $e$ is safely evaluated to a result of memory-type $\mu$. During the execution, the program may access the heap cells in $L$ and may deallocate those in $\mathcal{E}$. A set $\mathcal{V}$ of new names is introduced in the derivation and satisfies constraint $\mathcal{B}$. "`free` $v$ `when` $b$" has memory-type $\emptyset$ and deallocates $v$'s root cell when $b$ is true (FREE). A `Node`-expression introduces a new name $X$ for its new heap cell, and has a memory-type whose root is $X$ (NODE). For "`case` $v$ (`Node` $(x_1, x_2)$ `=>` $e_1$) (`Leaf =>` $e_2$)", when $v$ has memory-type $\emptyset$ which means that $v$ is a `Leaf`-value, the result of `case`-expression is the same as that of its `Leaf`-branch $e_2$ (LCASE), and when $v$ has a structured memory-type which means that $v$ is not a `Leaf`-value, the result of `case`-expression is the same as that of its `Node`-branch $e_1$ (NCASE). A function application has the result of its function body by replacing the formal parameter $A$, $\beta$, and $\beta_{\mathrm{ns}}$ by the actual argument $L$, $b$, and $b_{\mathrm{ns}}$, respectively (APP). For an expression "`let` $x$ `=` $e_1$ `in` $e_2$", its memory-type is that of $e_2$, it uses what $e_1$ or $e_2$ uses, it deallocates what $e_1$ or $e_2$ deallocates, and its constraint is, in addition to those of $e_1$ and $e_2$, that the heap cells freed by $e_1$ do not overlap with those used or freed by $e_2$ (LET).

$$\boxed{\Delta \vdash v : \mu \,\&\, \mathcal{C}}$$

$$\frac{\mathcal{C} \Rightarrow \emptyset \sqsubseteq \mu}{\Delta \vdash \texttt{Leaf} : \mu \,\&\, \mathcal{C}}\ (\textsc{Leaf}) \qquad \frac{id = x \text{ or } l \quad id \in \mathrm{dom}(\Delta) \qquad \mathcal{C} \Rightarrow \Delta(id) \sqsubseteq \mu}{\Delta \vdash id : \mu \,\&\, \mathcal{C}}\ (\textsc{Id})$$

$$\frac{\mathcal{C} \Rightarrow (\lambda\beta.\lambda\beta_{\mathrm{ns}}.\lambda A.\exists\mathcal{V}.\sigma \,\&\, \mathcal{C}) \sqsubseteq \mu \qquad \{y \mapsto \mu, x \mapsto A\} \vdash e : \exists\mathcal{V}.\sigma \,\&\, \mathcal{C}}{\Delta \vdash \texttt{fix}\ y\ [\beta, \beta_{\mathrm{ns}}]\ \lambda x.e : \mu \,\&\, \mathcal{C}'}\ (\textsc{Fun})$$

$$\boxed{\Delta \vdash e : \exists\mathcal{V}.\sigma \,\&\, \mathcal{C} \text{ where } \sigma = (\mathcal{B}, \mu, L, \mathcal{E})}$$ Every bound name is fresh: $\mathcal{V} \cap \mathrm{free}(\Delta) = \emptyset$.

$$\frac{\Delta \vdash v : \langle L, \mu_1, \mu_2\rangle \,\&\, \mathcal{C}}{\Delta \vdash \texttt{free}\ v\ \texttt{when}\ b : \exists\emptyset.\,(\emptyset, \emptyset, \emptyset, \{b \hookrightarrow L\}) \,\&\, \mathcal{C}}\ (\textsc{Free}) \qquad \frac{\Delta \vdash v : \mu \,\&\, \mathcal{C}}{\Delta \vdash v : \exists\emptyset.\,(\emptyset, \mu, \emptyset, \emptyset) \,\&\, \mathcal{C}}\ (\textsc{Value})$$

$$\frac{\Delta \vdash v_i : \mu_i \,\&\, \mathcal{C}}{\Delta \vdash \texttt{Node}\ (v_1, v_2) : \exists\{X\}.\,(\emptyset, \langle X, \mu_1, \mu_2\rangle, \emptyset, \emptyset) \,\&\, \mathcal{C}}\ (\textsc{Node})$$

$$\frac{\Delta \vdash v_1 : (\lambda\beta.\lambda\beta_{\mathrm{ns}}.\lambda A.\exists\mathcal{V}.\sigma \,\&\, \mathcal{C}) \,\&\, \mathcal{C}' \qquad \Delta \vdash v_2 : L \,\&\, \mathcal{C}' \quad \mathrm{free}(L) \cap \mathcal{V} = \emptyset \qquad \mathcal{S} \triangleq \{L/A,\ b/\beta,\ b_{\mathrm{ns}}/\beta_{\mathrm{ns}}\}}{\Delta \vdash v_1\ [b, b_{\mathrm{ns}}]\ v_2 : \exists\mathcal{V}.\,\mathcal{S}\sigma \,\&\, (\mathcal{S}\mathcal{C} \wedge \mathcal{C}')}\ (\textsc{App})$$

$$\frac{\Delta \vdash v : \emptyset \,\&\, \mathcal{C} \qquad \Delta \vdash e_2 : \exists\mathcal{V}.\sigma \,\&\, \mathcal{C}}{\Delta \vdash \texttt{case}\ v\ (\texttt{Node}\ (x_1, x_2)\ \texttt{=>}\ e_1)\ (\texttt{Leaf}\ \texttt{=>}\ e_2) : \exists\mathcal{V}.\sigma \,\&\, \mathcal{C}}\ (\textsc{Lcase})$$

$$\frac{\Delta \vdash e_1 : \exists\mathcal{V}_1.\sigma_1 \,\&\, \mathcal{C}_1 \text{ where } \sigma_1 = (\mathcal{B}, \mu, L, \mathcal{E}) \qquad \Delta \cup \{x \mapsto \mu\} \vdash e_2 : \exists\mathcal{V}_2.\sigma_2 \,\&\, \mathcal{C}_2 \qquad \mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset}{\Delta \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \exists\mathcal{V}_1 \cup \mathcal{V}_2.((\sigma_1 \,\&\, \mathcal{C}_1);\,(\sigma_2 \,\&\, \mathcal{C}_2))}\ (\textsc{Let})$$

$$\frac{\Delta \vdash v : \left\langle L', \mu_1, \mu_2\right\rangle \,\&\, \mathcal{C} \qquad \Delta \cup \{x_i \mapsto \mu_i\} \vdash e_1 : \exists\mathcal{V}.\,(\mathcal{B}, \mu, L, \mathcal{E}) \,\&\, \mathcal{C}}{\Delta \vdash \texttt{case}\ v\ (\texttt{Node}\ (x_1, x_2)\ \texttt{=>}\ e_1)(\texttt{Leaf}\ \texttt{=>}\ e_2) : \exists\mathcal{V}.\,(\mathcal{B}, \mu, L \mathbin{\dot\sqcup} L', \mathcal{E}) \,\&\, \mathcal{C}}\ (\textsc{Ncase})$$

where

$$(\sigma_1 \,\&\, \mathcal{C}_1);\,(\sigma_2 \,\&\, \mathcal{C}_2) \triangleq (\mathcal{B}_1 \cup \mathcal{B}_2, \mu_2, L_1 \mathbin{\dot\sqcup} L_2, \mathcal{E}_1 \cup \mathcal{E}_2) \,\&\, (\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge (\mathcal{E}_1 \# L_2) \wedge (\mathcal{E}_1 \# \mathcal{E}_2))$$

when $\sigma_i = (\mathcal{B}_i, \mu_i, L_i, \mathcal{E}_i)$.

Fig. 11. The memory-type system.

The memory-type system has five structural rules in Fig. 12. We can conclude with a greater result (WEAK). We can merge several $X_i$s into one name $X$ (MERGE). We can introduce new name $\pi$ by replacing $L_1$, $L_2$, and $L_3$ by $\pi$.root, $\pi$.left, and $\pi$.right, respectively, and recording that the collection of $\pi$.root, $\pi$.left, and $\pi$.right is equal to or smaller than the collection of $L_1$, $L_2$, and $L_3$ ($\pi$INT). We can introduce new name $R$ by replacing $L$ by $R$ in the judgment and recording that $R$ is equal to or smaller than $L$ (RINT). We can analyze a program by separating two cases of a variable in the environment. The

$\boxed{\Delta \vdash e : \exists \mathcal{V}.\, \sigma \,\&\, \mathcal{C} \text{ where } \sigma = (\mathcal{B}, \mu, L, \mathcal{E})}$ Every bound name is fresh: $\mathcal{V} \cap \text{free}(\Delta) = \emptyset$.

$$\frac{\Delta \vdash e : \exists \mathcal{V}'.\, \sigma' \,\&\, \mathcal{C}' \quad \mathcal{V}' \cap \text{free}(\mathcal{C}, \sigma) \subseteq \mathcal{V} \quad (\exists \mathcal{V}'.\, \sigma' \,\&\, \mathcal{C}') \sqsubseteq (\exists \mathcal{V}.\, \sigma \,\&\, \mathcal{C})}{\Delta \vdash e : \exists \mathcal{V}.\, \sigma \,\&\, \mathcal{C}}\ (\text{WEAK})$$

$$\frac{\Delta \vdash e : \exists \mathcal{V} \cup \{X_i\}.\, \mathcal{S}\sigma \,\&\, \mathcal{S}\mathcal{C} \quad \mathcal{S} \triangleq \left\{(\dot{\sqcup}_i X_i)/X\right\} \quad X_i \notin \text{free}(\sigma, \mathcal{C}) \quad X, X_i \notin \mathcal{V}}{\Delta \vdash e : \exists \mathcal{V} \cup \{X\}.\, \sigma \,\&\, \mathcal{C}}\ (\text{MERGE})$$

$$\frac{\Delta \vdash e : \exists \mathcal{V}.\, \mathcal{S}\sigma \,\&\, \mathcal{S}\mathcal{C} \quad \pi \notin \mathcal{V} \quad \mu \triangleq \langle L_1, L_2, L_3 \rangle \quad \text{PRECISE}(\mu) \quad \mathcal{S} \triangleq \{L_1/\pi.\text{root}, L_2/\pi.\text{left}, L_3/\pi.\text{right}\}}{\Delta \vdash e : \exists \mathcal{V} \cup \{\pi\}.\, (\sigma \cup \{\pi \mapsto \mathsf{collapse}(\mu)\}) \,\&\, \mathcal{C}}\ (\pi\text{INT})$$

$$\frac{\Delta \vdash e : \exists \mathcal{V}.\, \mathcal{S}\sigma \,\&\, \mathcal{S}\mathcal{C} \quad \mathcal{S} \triangleq \{L/R\} \quad R \notin \mathcal{V}}{\Delta \vdash e : \exists \mathcal{V} \cup \{R\}.\, (\sigma \cup \{R \mapsto L\}) \,\&\, \mathcal{C}}\ (\text{RINT})$$

$$\frac{\Delta \cup \{x \mapsto \langle \pi.\text{root}, \pi.\text{left}, \pi.\text{right} \rangle\} \vdash e : \exists \mathcal{V}.\, \sigma \,\&\, \mathcal{C} \quad \Delta \cup \{x \mapsto \emptyset\} \vdash e : \exists \mathcal{V}.\, \sigma \,\&\, \mathcal{C} \quad \pi \notin \mathcal{V}}{\Delta \cup \{x \mapsto \mu_{\mathsf{tree}}\} \vdash e : \exists \mathcal{V} \cup \{\pi\}.\, (\sigma \cup \{\pi \mapsto \mathsf{collapse}(\mu_{\mathsf{tree}})\}) \,\&\, \mathcal{C}}\ (\text{PRUNE})$$

where

$$\sigma_1 \cup \mathcal{B} \triangleq (\mathcal{B}_1 \cup \mathcal{B}, \mu_1, L_1, \mathcal{E}_1)$$

$(\exists \mathcal{V}_1.\, \sigma_1 \,\&\, \mathcal{C}_1) \sqsubseteq (\exists \mathcal{V}_2.\, \sigma_2 \,\&\, \mathcal{C}_2)$ iff
$\mathcal{V}_1 \supseteq \mathcal{V}_2,\ \mathcal{B}_1 \Rightarrow \mathcal{B}_2,$ and $\mathcal{B}_1 \wedge \mathcal{C}_2 \Rightarrow \mathcal{C}_1 \wedge (\mu_1 \sqsubseteq \mu_2) \wedge (L_1 \sqsubseteq_{\mathsf{set}} L_2) \wedge (\mathcal{E}_1 \sqsubseteq \mathcal{E}_2)$

when $\sigma_i = (\mathcal{B}_i, \mu_i, L_i, \mathcal{E}_i)$.

Fig. 12. The structural rules of the memory-type system.

separation is when the variable has a `Leaf`-value or not. The result is the one where both cases agree (PRUNE).

The memory-type system for a state is defined in Fig. 13. A state $(e, h, f, k)$ is well-typed when each component is well-typed, the constraints $(\mathcal{C}_1 \wedge \mathcal{C}_2)$ of expression $e$ and continuation $k$ are satisfied, and it is safe to sequentially evaluate $e$ and $k$ when the heap cells of locations $f$ are freed (STATE). Note that the side conditions make sure that the freed heap cells of locations $f$ should be neither used nor freed by $e$ or $k$ $(\mathcal{C}_{(0,1)} \wedge \mathcal{C}_{(0,2)})$ and the heap cells freed by $e$ should be neither used nor freed by $k$ $(\mathcal{C}_{(1,2)})$. In rules (NIL) and (CONT), we use a special identifier $\bullet$ for the argument of a continuation.

## 5.2. The memory-type system is sound

We prove the soundness of the memory-type system by the syntactic approach [27]. The key propositions are, as usual:

$\boxed{\Delta \vdash k : \exists \mathcal{V}. \sigma \,\&\, \mathcal{C} \ \mathit{where}\ \sigma = (\mathcal{B}, \mu, L, \mathcal{E})}$ Every bound name is fresh: $\mathcal{V} \cap \mathrm{free}(\Delta) = \emptyset$.

$$\frac{}{\begin{array}{l}\Delta \cup \{\bullet \mapsto \mu\} \vdash \epsilon : \\ \exists \emptyset. (\emptyset, \emptyset, \emptyset, \emptyset) \,\&\, \mathcal{C}\end{array}}\ (\text{NIL}) \qquad \frac{\begin{array}{l}\Delta \cup \{x \mapsto \mu\} \vdash e : \exists \mathcal{V}_1. \sigma_1 \,\&\, \mathcal{C}_1 \\ \qquad \text{where } \sigma_1 = (\mathcal{B}, \mu_1, L, \mathcal{E}) \\ \Delta \cup \{\bullet \mapsto \mu_1\} \vdash k : \exists \mathcal{V}_2. \sigma_2 \,\&\, \mathcal{C}_2 \\ \mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset\end{array}}{\begin{array}{l}\Delta \cup \{\bullet \mapsto \mu\} \vdash (x, e) \cdot k : \\ \exists \mathcal{V}_1 \cup \mathcal{V}_2. ((\sigma_1 \,\&\, \mathcal{C}_1); (\sigma_2 \,\&\, \mathcal{C}_2))\end{array}}\ (\text{CONT})$$

$\boxed{\vdash h : \Delta}$

$$\frac{\begin{array}{l}\Delta \triangleq \{l_1 \mapsto \langle X_1, \mu_{(1,1)}, \mu_{(1,2)} \rangle, \ldots, l_n \mapsto \langle X_n, \mu_{(n,1)}, \mu_{(n,2)} \rangle\} \\ \forall i \neq j. X_i \neq X_j \quad \forall i, j. \mu_{(i,j)} \triangleq \begin{cases} \Delta(l), & \text{when } a_{(i,j)} = l \\ \emptyset, & \text{when } a_{(i,j)} = \texttt{Leaf} \end{cases}\end{array}}{\vdash \{l_1 \mapsto (a_{(1,1)}, a_{(1,2)}), \ldots, l_n \mapsto (a_{(n,1)}, a_{(n,2)})\} : \Delta}\ (\text{HEAP})$$

$\boxed{\Delta \vdash f : \mathcal{E}}$

$$\frac{\forall l_i \in f. \Delta(l_i) = \langle X_i, \mu_i, \mu_i' \rangle}{\Delta \vdash f : \{\text{true} \hookrightarrow X_i \mid l_i \in f\}}\ (\text{FREED})$$

$\boxed{\vdash (e, h, f, k)}$

$$\frac{\begin{array}{l}\vdash h : \Delta \quad \Delta \vdash f : \mathcal{E}_0 \\ \Delta \vdash e : \exists \mathcal{V}_1. \sigma_1 \,\&\, \mathcal{C}_1 \text{ where } \sigma_1 = (\mathcal{B}_1, \mu_1, L_1, \mathcal{E}_1) \\ \Delta \cup \{\bullet \mapsto \mu_1\} \vdash k : \exists \mathcal{V}_2. \sigma_2 \,\&\, \mathcal{C}_2 \text{ where } \sigma_2 = (\mathcal{B}_2, \mu_2, L_2, \mathcal{E}_2) \\ \mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset \\ \mathcal{B}_1 \wedge \mathcal{B}_2 \Rightarrow \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_{(0,1)} \wedge \mathcal{C}_{(1,2)} \wedge \mathcal{C}_{(0,2)} \text{ where } \mathcal{C}_{(i,j)} = \mathcal{E}_i \# L_j \wedge \mathcal{E}_i \# \mathcal{E}_j\end{array}}{\vdash (e, h, f, k)}\ (\text{STATE})$$

where
$$(\sigma_1 \,\&\, \mathcal{C}_1); (\sigma_2 \,\&\, \mathcal{C}_2) \triangleq \begin{array}{l}(\mathcal{B}_1 \cup \mathcal{B}_2, \mu_2, L_1 \dot{\cup} L_2, \mathcal{E}_1 \cup \mathcal{E}_2) \\ \,\&\, (\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge (\mathcal{E}_1 \# L_2) \wedge (\mathcal{E}_1 \# \mathcal{E}_2))\end{array}$$
when $\sigma_i = (\mathcal{B}_i, \mu_i, L_i, \mathcal{E}_i)$.

Fig. 13. The memory-type system for states.

- **subject reduction**: if a well-typed state has a transition, the next state is also well-typed (Proposition 1); and
- **progress**: there exists a transition from the well-typed state, or the well-typed state is final (Proposition 2).

In order to achieve the above two key propositions, we need to establish several lemmas:

- we can rename the names in our judgments (Lemma 1);
- we can substitute multiset formulas for free names, or boolean expressions for free boolean variables in our judgments (Lemma 2);

- we can substitute values for program variables in our judgments when their memory-types are the same (Lemma 3); and
- our typing derivation is monotonic (Lemma 4).

**Lemma 1** (*Fresh Names*). *For a memory-type environment $\Delta$, an expression $e$, a set $\mathcal{V}$ of names, a result $\sigma$, and a constraint $\mathcal{C}$, if $\Delta \vdash e : \exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C}$, then for a substitution $\mathcal{S} = \left\{V'/V\right\}$ with $V'$ being a fresh name of the same kind as $V$, $\mathcal{S}\Delta \vdash e : \exists\{\mathcal{S}V \mid V \in \mathcal{V}\}.\,\mathcal{S}\sigma \,\&\, \mathcal{S}\mathcal{C}$.*

**Proof.** By structural induction on the derivation trees. $\square$

We can apply a substitution to judgments only when the substitution respects the conditions of good environments. Note that a substitution can violate the good environment conditions; for instance, $\pi$.root and $\pi$.left are disjoint in a good environment whereas $\mathcal{S}(\pi.\text{root})$ and $\mathcal{S}(\pi.\text{left})$ can overlap each other when $\mathcal{S} = \{X/\pi.\text{root}, X/\pi.\text{left}\}$. The side conditions of substitution (b)–(d) in Lemma 2 are for preserving the conditions of good environments.

**Lemma 2** (*Type Replacement*). *For constraints $\mathcal{C}_1, \mathcal{C}_2$, and $\mathcal{C}$, a memory-type environment $\Delta$, a value $v$, an expression $e$, a memory-type $\mu$, a set $\mathcal{V}$ of names, and a result $\sigma$, the following are true:*

(1) *if $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$, then $\mathcal{S}\mathcal{C}_1 \Rightarrow \mathcal{S}\mathcal{C}_2$;*
(2) *if $\Delta \vdash v : \mu \,\&\, \mathcal{C}$, then $\mathcal{S}\Delta \vdash v : \mathcal{S}\mu \,\&\, \mathcal{S}\mathcal{C}$; and*
(3) *if $\Delta \vdash e : \exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C}$ holds and $\mathcal{V} \cap \text{free}(\mathcal{S}) = \emptyset$, then $\mathcal{S}\Delta \vdash \mathcal{S}e : \exists\mathcal{V}.\,\mathcal{S}\sigma \,\&\, \mathcal{S}\mathcal{C}$ holds with the same size of derivation tree; and the same lemma holds for continuation $k$,*

*when $\mathcal{S}$ is either*

(a) *$\{L/R\}$;*
(b) *$\{L_1/\pi.\text{root}, L_2/\pi.\text{left}, L_3/\pi.\text{right}\}$ where $\textsc{precise}(\langle L_1, L_2, L_3\rangle)$ holds;*
(c) *$\{L/X\}$ where $L$ consists of fresh $X_i$s and $\textsc{set}(L)$ holds;*
(d) *$\{L/A\}$ where $L$ consists of fresh $X_i$s and $A_i$s; or*
(e) *$\{b_1/\beta_1, \ldots, b_n/\beta_n\}$.*

**Proof.** The proof is in [11]. $\square$

We can replace a variable in judgments by a value when the variable and the value have the same memory-type. The exception is that the memory-type is not *precise*: a memory-type $\mu$ is not precise if and only if $\mu$ is structured and its root and left/right sub-tree can be overlapped; for instance, $\langle X_1, X_1, X_2\rangle$ is not precise because the root part $X_1$ and the left sub-tree $X_1$ are overlapped. This exception is because we only have a pruning rule PRUNE restricted for a variable: after replacing a variable by a value, since we cannot apply rule PRUNE in the same way, we may not derive the same judgment.

**Lemma 3** (*Term Replacement*). *For a memory-type environment $\Delta$, a variable $x$, values $v$ and $v'$, an expression $e$, memory-types $\mu$ and $\mu'$, a constraint $\mathcal{C}$, a set $\mathcal{V}$ of names, and a result $\sigma$, the following are true:*

(1) *If $\Delta \cup \{x \mapsto \mu\} \vdash v' : \mu' \,\&\, \mathcal{C}$ and $\Delta \vdash v : \mu \,\&\, \mathcal{C}$, then $\Delta \vdash v'\,\{v/x\} : \mu' \,\&\, \mathcal{C}$.*

(2) *If $\Delta \cup \{x \mapsto \mu\} \vdash e : \exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C}$ and $\Delta \vdash v : \mu \,\&\, \mathcal{C}$, then $\Delta \vdash e\{v/x\} : \exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C}$ unless $v$ is a* tree*-typed identifier and* $\mathrm{PRECISE}(\mu)$ *does not hold.*

**Proof.** The proof is in [11]. □

Our typing derivation is monotonic. When a judgment holds with a memory-type environment $\Delta$, by using a stronger one than $\Delta$, we can derive another judgment whose result is stronger than the original one.

**Lemma 4** (*Monotonicity*). *For a memory-type environment $\Delta$, a value $v$, an expression $e$, a memory-type $\mu$, a constraint $\mathcal{C}$, a set $\mathcal{V}$ of names, and a result $\sigma$, the following are true:*

(1) *If $\Delta \vdash v : \mu \,\&\, \mathcal{C}$ and $\mathcal{C} \Rightarrow \Delta' \sqsubseteq \Delta$, there exists a memory-type $\mu'$ such that $\Delta' \vdash v : \mu' \,\&\, \mathcal{C}$ and $\mathcal{C} \Rightarrow \mu' \sqsubseteq \mu$.*
(2) *If*
 (a) $\mathcal{C} \Rightarrow \Delta' \sqsubseteq \Delta$,
 (b) $\Delta \vdash e : \exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C}$, *and*
 (c) $\mathcal{V} \cap \mathrm{free}(\Delta') = \emptyset$,
 *then there exist a result $\sigma'$ and a constraint $\mathcal{C}'$ such that $\Delta' \vdash e : \exists\mathcal{V}.\,\sigma' \,\&\, \mathcal{C}'$ and $(\exists\mathcal{V}.\,\sigma' \,\&\, \mathcal{C}') \sqsubseteq (\exists\mathcal{V}.\,\sigma \,\&\, \mathcal{C})$. Moreover, the same lemma holds for continuation $k$,*

*where $\mathcal{C} \Rightarrow \Delta' \sqsubseteq \Delta$ if and only if $\mathrm{dom}(\Delta') \supseteq \mathrm{dom}(\Delta)$ and for all $id \in \mathrm{dom}(\Delta)$, $\mathcal{C} \Rightarrow \Delta'(id) \sqsubseteq \Delta(id)$.*

**Proof.** The proof is in [11]. □

**Proposition 1** (*Subject Reduction*). *For states $(e, h, f, k)$ and $(e', h', f', k')$, if $\vdash (e, h, f, k)$ and $(e, h, f, k) \rightsquigarrow (e', h', f', k')$, we have $\vdash (e', h', f', k')$.*

**Proof.** For each transition $(e, h, f, k) \rightsquigarrow (e', h', f', k')$ in Fig. 1, we derive $\vdash (e', h', f', k')$ from $\vdash (e, h, f, k)$. By (STATE),

$$\vdash h : \Delta, \tag{1}$$
$$\Delta \vdash f : \mathcal{E}_0, \tag{2}$$
$$\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset, \tag{3}$$
$$\Delta \vdash e : \exists\mathcal{V}_1.\,\sigma_1 \,\&\, \mathcal{C}_1 \text{ where } \sigma_1 = (\mathcal{B}_1, \mu_1, L_1, \mathcal{E}_1), \tag{4}$$
$$\Delta \cup \{\bullet \mapsto \mu_1\} \vdash k : \exists\mathcal{V}_2.\,\sigma_2 \,\&\, \mathcal{C}_2 \text{ where } \sigma_2 = (\mathcal{B}_2, \mu_2, L_2, \mathcal{E}_2), \text{ and} \tag{5}$$
$$(\mathcal{B}_1 \wedge \mathcal{B}_2) \Rightarrow \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_{(0,1)} \wedge \mathcal{C}_{(1,2)} \wedge \mathcal{C}_{(0,2)} \tag{6}$$

where $\mathcal{C}_{(i,j)} = \mathcal{E}_i \# L_j \wedge \mathcal{E}_i \# \mathcal{E}_j$. In order to avoid the case that (4) ends with the structural rules (WEAK), (MERGE), (RINT), ($\pi$INT), and (PRUNE), we first prove that there is another derivation tree for $\vdash (e, h, f, k)$ where (4) does not end with the structural rules. We prove it by induction on the size of the derivation tree of (4):

- **case** (WEAK): : The assumption is that (4) is derived by (WEAK); that is, there exist $\mathcal{V}_1'$, $\mathcal{C}_1'$, and $\sigma_1'$ such that

$$\Delta \vdash e : \exists\mathcal{V}_1'.\,(\mathcal{B}_1', \mu_1', L_1', \mathcal{E}_1') \,\&\, \mathcal{C}_1', \tag{7}$$
$$\mathcal{V}_1' \cap \mathrm{free}(\sigma_1, \mathcal{C}_1) \subseteq \mathcal{V}_1, \tag{8}$$

$$\mathcal{V}_1 \subseteq \mathcal{V}_1', \tag{9}$$
$$\mathcal{B}_1' \Rightarrow \mathcal{B}_1, \text{ and} \tag{10}$$
$$\mathcal{B}_1' \wedge \mathcal{C}_1 \Rightarrow \mathcal{C}_1' \wedge (\mu_1' \sqsubseteq \mu_1) \wedge (L_1' \sqsubseteq_{\mathsf{set}} L_1) \wedge (\mathcal{E}_1' \sqsubseteq \mathcal{E}_1). \tag{11}$$

We can assume that $\mathcal{V}_1' \setminus \mathcal{V}_1$ are fresh by Lemma 1 and (8). Then (3) and (9) imply that

$$\mathcal{V}_1' \cap \mathcal{V}_2 = \emptyset. \tag{12}$$

(5) implies that

$$\Delta \cup \{\bullet \mapsto \mu_1\} \vdash k : \exists \mathcal{V}_2.\, \sigma_2 \,\&\, (\mathcal{C}_2 \wedge \mathcal{B}_1' \wedge \mathcal{B}_2). \tag{13}$$

because

- when $k = \epsilon$, $\Delta \cup \{\bullet \mapsto \mu_1\} \vdash \epsilon : \exists\emptyset.\, (\emptyset, \emptyset, \emptyset, \emptyset) \,\&\, \mathcal{C}$ for any $\mathcal{C}$, and
- when $k = (x, e) \cdot k'$, (5) has sub-judgment $\Delta \cup \{x \mapsto \mu_1\} \vdash e : \exists\mathcal{V}.\, \sigma \,\&\, \mathcal{C}$ for some $\mathcal{V}$, $\sigma$ and $\mathcal{C}$. By (WEAK), $\Delta \cup \{x \mapsto \mu_1\} \vdash e : \exists\mathcal{V}.\, \sigma \,\&\, (\mathcal{C} \wedge \mathcal{B}_1' \wedge \mathcal{B}_2)$. Then by (CONT), we achieve (13).

(6), (10) and (11) imply that $\mathcal{B}_1' \wedge \mathcal{B}_2 \Rightarrow \mu_1' \sqsubseteq \mu_1$. Then $\mathcal{B}_1' \wedge \mathcal{B}_2 \wedge \mathcal{C}_2 \Rightarrow \Delta \cup \left\{\bullet \mapsto \mu_1'\right\} \sqsubseteq \Delta \cup \{\bullet \mapsto \mu_1\}$. (12) implies that $\mathrm{free}(\mu_1') \cap \mathcal{V}_2 = \emptyset$ because $\mathrm{free}(\mu_1') \subseteq \mathcal{V}_1'$. Then by Lemma 4, (13) implies that there exist $\mathcal{B}_2'$, $\mu_2'$, $L_2'$, $\mathcal{E}_2'$, and $\mathcal{C}_2'$ such that

$$\Delta \cup \left\{\bullet \mapsto \mu_1'\right\} \vdash k : \exists\mathcal{V}_2.\, (\mathcal{B}_2', \mu_2', L_2', \mathcal{E}_2') \,\&\, \mathcal{C}_2', \tag{14}$$
$$\mathcal{B}_2' \Rightarrow \mathcal{B}_2, \text{ and} \tag{15}$$
$$\mathcal{B}_1' \wedge \mathcal{B}_2 \wedge \mathcal{C}_2 \Rightarrow \mathcal{C}_2' \wedge (\mu_2' \sqsubseteq \mu_2) \wedge (L_2' \sqsubseteq_{\mathsf{set}} L_2) \wedge (\mathcal{E}_2' \sqsubseteq \mathcal{E}_2). \tag{16}$$

(6), (10), and (15) imply that

$$\mathcal{B}_1' \wedge \mathcal{B}_2' \Rightarrow \mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \mathcal{C}_1 \wedge \mathcal{C}_2. \tag{17}$$

(11), (16), and (17) imply that

$$\begin{aligned}&\mathcal{B}_1' \wedge \mathcal{B}_2' \Rightarrow \\ &\quad \mathcal{C}_1' \wedge \mathcal{C}_2' \wedge (L_1' \sqsubseteq_{\mathsf{set}} L_1) \wedge (\mathcal{E}_1' \sqsubseteq \mathcal{E}_1) \wedge (L_2' \sqsubseteq_{\mathsf{set}} L_2) \wedge (\mathcal{E}_2' \sqsubseteq \mathcal{E}_2).\end{aligned} \tag{18}$$

(6) and (17) imply that

$$\mathcal{B}_1' \wedge \mathcal{B}_2' \Rightarrow \mathcal{E}_0 \# L_1 \wedge \mathcal{E}_0 \# \mathcal{E}_1 \wedge \mathcal{E}_0 \# L_2 \wedge \mathcal{E}_0 \# \mathcal{E}_2 \wedge \mathcal{E}_1 \# L_2 \wedge \mathcal{E}_1 \# \mathcal{E}_2. \tag{19}$$

(18) and (19) imply that

$$\mathcal{B}_1' \wedge \mathcal{B}_2' \Rightarrow \mathcal{E}_0 \# L_1' \wedge \mathcal{E}_0 \# \mathcal{E}_1' \wedge \mathcal{E}_0 \# L_2' \wedge \mathcal{E}_0 \# \mathcal{E}_2' \wedge \mathcal{E}_1' \# L_2' \wedge \mathcal{E}_1' \# \mathcal{E}_2'. \tag{20}$$

By (STATE), (1), (2), (7), (12), (14), (18), and (20) imply that $\vdash (e, h, f, k)$.

- **case** (RINT): The assumption is that (4) is derived by (RINT); that is, when $\mathcal{S} = \{L/R\}$, $\mathcal{V}_1 = \mathcal{V}_1' \cup \{R\}$, and $\sigma_1 = \sigma_1' \cup \{R \mapsto L\}$,

$$\Delta \vdash e : \exists\mathcal{V}_1'.\, \mathcal{S}\sigma_1' \,\&\, \mathcal{S}\mathcal{C}_1. \tag{21}$$

By Lemma 2, we can apply $\mathcal{S}$ to (5) and (6):

$$\mathcal{S}\Delta \cup \{\bullet \mapsto \mathcal{S}\mu_1\} \vdash k : \exists\mathcal{V}_2.\, \mathcal{S}\sigma_2 \,\&\, \mathcal{S}\mathcal{C}_2, \text{ and} \tag{22}$$
$$(\mathcal{S}\mathcal{B}_1 \wedge \mathcal{S}\mathcal{B}_2) \Rightarrow \mathcal{S}\mathcal{C}_1 \wedge \mathcal{S}\mathcal{C}_2 \wedge \mathcal{S}\mathcal{C}_{(0,1)} \wedge \mathcal{S}\mathcal{C}_{(1,2)} \wedge \mathcal{S}\mathcal{C}_{(0,2)}. \tag{23}$$

Note that since $R$ does not appear in $\Delta$ and $\mathcal{E}_0$, $\mathcal{S}\Delta = \Delta$ and $\mathcal{S}\mathcal{E}_0 = \mathcal{E}_0$, that is, $\mathcal{S}\mathcal{C}_{(0,i)} = \mathcal{E}_0\#\mathcal{S}L_i \wedge \mathcal{E}_0\#\mathcal{S}\mathcal{E}_i$. Then by (STATE), (1)–(3) and (21)–(23) imply that $\vdash (e, h, f, k)$.

- **case** ($\pi$INT) and (MERGE): These cases are proved similarly to the case (RINT).
- **case** (PRUNE): (4) cannot be derived by (PRUNE) because $\mathrm{dom}(\Delta)$ has only locations.

We prove by case analysis with the assumption that (4) does not end with the structural rules.

- **case** $(\texttt{free}\, l \;\texttt{when}\, b, h, f, k) \rightsquigarrow (\texttt{Leaf}, h, f \cup \{l\}, k)$ when $l \in \mathrm{dom}(h)$, $l \notin f$, and $b \Leftrightarrow \mathrm{true}$.: In this case, (4) is

$$\Delta \vdash \texttt{free}\, l \;\texttt{when}\, b : \exists\emptyset.\,(\emptyset, \emptyset, \emptyset, \{\mathrm{true} \hookrightarrow L\})\,\&\,\mathcal{C}_1.$$

By (FREE), $\Delta \vdash l : \langle L, \mu_1, \mu_2\rangle\,\&\,\mathcal{C}_1$ for some $\mu_1$ and $\mu_2$. By (HEAP), $\Delta(l) = \left\langle X, \mu_1', \mu_2'\right\rangle$ for some $X$, $\mu_1'$, and $\mu_2'$, and by (ID), $\mathcal{C}_1 \Rightarrow X \sqsubseteq L$. Since (6) implies that $\emptyset \Rightarrow \mathcal{C}_1$, we have $X \sqsubseteq L$. By (FREED), (2) implies that

$$\Delta \vdash f \cup \{l\} : \mathcal{E}_0 \cup \{\mathrm{true} \hookrightarrow X\}\,. \tag{24}$$

By (LEAF) and (VALUE),

$$\Delta \vdash \texttt{Leaf} : \exists\emptyset.\,(\emptyset, \emptyset, \emptyset, \emptyset)\,\&\,\mathcal{C}_1. \tag{25}$$

Since $\mathcal{E}_0 \cup \{\mathrm{true} \hookrightarrow X\} \sqsubseteq \mathcal{E}_0 \cup \mathcal{E}_1$, (6) implies that

$$\mathcal{B}_1 \wedge \mathcal{B}_2 \Rightarrow (\mathcal{E}_0 \cup \{\mathrm{true} \hookrightarrow X\})\#L_2 \wedge (\mathcal{E}_0 \cup \{\mathrm{true} \hookrightarrow X\})\#\mathcal{E}_2. \tag{26}$$

Therefore by (STATE), (1), (3), (5), (6), and (24)–(26) imply that

$$\vdash (\texttt{Leaf}, h, f \cup \{l\}, k).$$

- **case** $(e, h, f, k) \rightsquigarrow (e_1\,\{a_1/x_1, a_2/x_2\}, h, f, k)$ when $h(l) = (a_1, a_2)$, $l \notin f$, and $e = \texttt{case}\, l\; (\texttt{Node}\,(x_1, x_2) \texttt{=>}\, e_1)\; (\texttt{Leaf} \texttt{=>}\, e_2)$: (4) is

$$\Delta \vdash e : \exists\mathcal{V}_1.\,(\mathcal{B}_1, \mu_1, L_1, \mathcal{E}_1)\,\&\,\mathcal{C}_1. \tag{27}$$

By (HEAP), $\Delta(l) = \langle X, \mu_1, \mu_2\rangle$ for some $X$, and precise $\mu_1$ and $\mu_2$. Since it is impossible to have $\mathcal{C} \Rightarrow \Delta(l) \sqsubseteq \emptyset$ for any $\mathcal{C}$, (27) is derived by (NCASE); that is,

$$\Delta \cup \left\{x_i \mapsto \mu_i'\right\} \vdash e_1 : \exists\mathcal{V}_1.\,(\mathcal{B}_1, \mu_1, L_1', \mathcal{E}_1)\,\&\,\mathcal{C}_1, \text{ and} \tag{28}$$

$$\Delta \vdash l : \left\langle L, \mu_1', \mu_2'\right\rangle\&\,\mathcal{C}_1, \tag{29}$$

where $L_1' \dot\sqcup L = L_1$. Since $\Delta(l) = \langle X, \mu_1, \mu_2\rangle$, by (ID), (29) implies that $\mathcal{C}_1 \Rightarrow \mu_i \sqsubseteq \mu_1'$. By Lemma 4, (28) implies that

$$\Delta \cup \{x_i \mapsto \mu_i\} \vdash e_1 : \exists\mathcal{V}_1.\,(\mathcal{B}_1, \mu_1, L_1', \mathcal{E}_1)\,\&\,\mathcal{C}_1. \tag{30}$$

By (HEAP), (ID), and (LEAF), $\Delta \vdash a_i : \mu_i\,\&\,\mathcal{C}_1$. Then by Lemma 3, (30) implies that $\Delta \vdash e_1\,\{a_1/x_1, a_2/x_2\} : \exists\mathcal{V}_1.\,(\mathcal{B}_1, \mu_1, L_1', \mathcal{E}_1)\,\&\,\mathcal{C}_1$. By (WEAK),

$$\Delta \vdash e_1\,\{a_1/x_1, a_2/x_2\} : \exists\mathcal{V}_1.\,(\mathcal{B}_1, \mu_1, L_1, \mathcal{E}_1)\,\&\,\mathcal{C}_1. \tag{31}$$

Then by (STATE), (1)–(3), (5), (6), and (30) imply that

$$\vdash (e_1\,\{a_1/x_1, a_2/x_2\}, h, f, k).$$

- **case** $(F\ [b, b_{\text{ns}}]\ v, h, f, k) \rightsquigarrow (e\,\{b/\beta, b_{\text{ns}}/\beta_{\text{ns}}\}\,\{F/y\}\,\{v/x\}\,, h, f, k)$ where $F = \texttt{fix}\ y\ [\beta_1, \beta_2]\ \lambda x.e.$: (4) is

  $$\Delta \vdash F\ [b, b_{\text{ns}}]\ v : \exists\mathcal{V}.\,\mathcal{S}\sigma\ \&\ (\mathcal{S}\mathcal{C} \wedge \mathcal{C}').$$

  By (APP), when $\mathcal{S} = \{L/A, b/\beta, b_{\text{ns}}/\beta_{\text{ns}}\}$,

  $$\Delta \vdash \texttt{fix}\ y\ [\beta_1, \beta_2]\ \lambda x.e : \mu\,\&\,\mathcal{C}' \text{ where } \mu = \lambda\beta.\lambda\beta_{\text{ns}}.\lambda A.\exists\mathcal{V}.\,\sigma\,\&\,\mathcal{C}, \tag{32}$$
  $$\Delta \vdash v : L\,\&\,\mathcal{C}', \text{ and} \tag{33}$$
  $$\text{free}(L) \cap \mathcal{V} = \emptyset. \tag{34}$$

  By (FUN), (32) implies that $\{y \mapsto \mu, x \mapsto A\} \vdash e : \exists\mathcal{V}.\,\sigma\,\&\,\mathcal{C}$. By (34) and Lemma 2, applying $\mathcal{S}$ to the judgment,

  $$\{y \mapsto \mu, x \mapsto L\} \vdash e\,\{b/\beta, b_{\text{ns}}/\beta_{\text{ns}}\} : \exists\mathcal{V}.\,\mathcal{S}\sigma\,\&\,\mathcal{S}\mathcal{C}.$$

  By Lemma 4,

  $$\Delta \cup \{y \mapsto \mu, x \mapsto L\} \vdash e\,\{b/\beta, b_{\text{ns}}/\beta_{\text{ns}}\} : \exists\mathcal{V}.\,\mathcal{S}\sigma\,\&\,\mathcal{S}\mathcal{C}.$$

  By (32), (33), and Lemma 3,

  $$\Delta \vdash e\,\{b_1/\beta_1, b_2/\beta_2\}\,\{F/y\}\,\{v/x\} : \exists\mathcal{V}.\,\mathcal{S}\sigma\ \&\ (\mathcal{S}\mathcal{C} \wedge \mathcal{C}'). \tag{35}$$

  By (STATE), (1)–(3), (5), (6), and (35) imply that

  $$\vdash (e\,\{b_1/\beta_1, b_2/\beta_2\}\,\{F/y\}\,\{v/x\}\,, h, f, k).$$

The proofs for other cases are in [11]. □

**Proposition 2** (*Progress*). *If a state $(e, h, f, k)$ is well-typed (i.e., $\vdash (e, h, f, k)$), then $(e, h, f, k)$ is final (i.e., $e$ is a value and $k$ is an empty continuation $\epsilon$), or there exists a transition $(e, h, f, k) \rightsquigarrow (e', h', f', k')$ for some $(e', h', f', k')$.*

**Proof.** We consider only the cases of memory errors; non-closed or ill-typed states in the ordinary type system are straightforwardly rejected by our memory-type system.

- **case** $(\texttt{free}\ l\ \texttt{when}\ b, h, f, k)$ when $b \Leftrightarrow$ true, $l \in f$, and $l \in \text{dom}(h)$: Assume for a contradiction that $\vdash (\texttt{free}\ l\ \texttt{when}\ b, h, f, k)$. By (STATE),

  $$\vdash h : \Delta, \tag{36}$$
  $$\Delta \vdash f : \mathcal{E}_0, \tag{37}$$
  $$\Delta \vdash \texttt{free}\ l\ \texttt{when}\ b : \exists\mathcal{V}.\,\sigma\,\&\,\mathcal{C} \text{ where } \sigma = (\mathcal{B}, \mu, L, \mathcal{E}), \text{ and} \tag{38}$$
  $$\mathcal{B} \Rightarrow \mathcal{C} \wedge (\mathcal{E}_0 \# \mathcal{E}). \tag{39}$$

  As we did when we proved Proposition 1, we can assume that (38) does not end with the structural rules; that is, by (FREE), $\mathcal{B} = \emptyset$, $\mathcal{E} = \left\{b \hookrightarrow L'\right\}$, and

  $$\Delta \vdash l : \left\langle L', \mu_1, \mu_2 \right\rangle \& \,\mathcal{C}$$

  for some $\mu_1$ and $\mu_2$. By (ID), $\mathcal{C} \Rightarrow \Delta(l) \sqsubseteq \left\langle L', \mu_1, \mu_2 \right\rangle$. By (HEAP) and (36), $\Delta(l) = \left\langle X, \mu_1', \mu_2' \right\rangle$ for some $X$, $\mu_1'$, and $\mu_2'$. Since $\mathcal{B} = \emptyset$, $\mathcal{B} \Rightarrow \mathcal{C}$, $\mathcal{C} \Rightarrow X \sqsubseteq L'$, and $\mathcal{E} = \left\{b \hookrightarrow L'\right\}$, and $b \Leftrightarrow$ true, we can conclude that (39) implies that $\mathcal{E}_0\#\{\text{true} \hookrightarrow X\}$

holds. By (FREED) and (37), $\mathcal{E}_0$ has $\{\text{true} \hookrightarrow X\}$. Then our conclusion becomes $\{\text{true} \hookrightarrow X\}\#\{\text{true} \hookrightarrow X\}$ which does not hold.

- **case** $(\texttt{case}\ l\ (\texttt{Node}\ (x_1, x_2) \texttt{=>} e_1)\ (\texttt{Leaf} \texttt{=>} e_2), h, f, k)$ when $l \in f$: Assume for a contradiction that $\vdash (\texttt{case}\ l\ (\texttt{Node}\ (x_1, x_2) \texttt{=>} e_1)\ (\texttt{Leaf} \texttt{=>} e_2), h, f, k)$. By (STATE),

$$\vdash h : \Delta, \tag{40}$$

$$\Delta \vdash f : \mathcal{E}_0, \tag{41}$$

$$\Delta \vdash \texttt{case}\ l\ (\texttt{Node}\ (x_1, x_2) \texttt{=>} e_1)\ (\texttt{Leaf} \texttt{=>} e_2) : \exists \mathcal{V}.(\mathcal{B}, \mu, L, \mathcal{E})\ \&\ \mathcal{C}, \tag{42}$$

$$\mathcal{B} \Rightarrow \mathcal{C} \wedge (\mathcal{E}_0\#\{\text{true} \hookrightarrow L\}). \tag{43}$$

We can assume that (42) is derived by (NCASE); that is,

$$\Delta \vdash l : \langle L, \mu_1, \mu_2 \rangle$$

for some $\mu_1$ and $\mu_2$. By (ID), $\mathcal{C} \Rightarrow \Delta(l) \sqsubseteq \langle L, \mu_1, \mu_2 \rangle$. By (HEAP) and (40), $\Delta(l) = \langle X, \mu_1', \mu_2' \rangle$ for some $X$, $\mu_1'$, and $\mu_2'$. Since $\mathcal{B} \Rightarrow \mathcal{C}$ and $\mathcal{C} \Rightarrow X \sqsubseteq L$, we can conclude that (43) implies that $\mathcal{B} \Rightarrow \mathcal{E}_0\#\{\text{true} \hookrightarrow X\}$. By (FREED) and (41), $\mathcal{E}_0$ has $\{\text{true} \hookrightarrow X\}$. Then our conclusion becomes $\mathcal{B} \Rightarrow \{\text{true} \hookrightarrow X\}\#\{\text{true} \hookrightarrow X\}$; that is, $\mathcal{B} \Rightarrow X\#X$ which does not hold. □

**Theorem 1** (*Memory-Type Soundness*). *If a state $(e, h, f, k)$ is well-typed in the memory-type system (i.e., $\vdash (e, h, f, k)$), then $(e, h, f, k)$ does not go to a stuck state: $(e, h, f, k) \leadsto^* (v, h', f', \epsilon)$ for some $v$, $h'$, and $f'$, or a transition from $(e, h, f, k)$ does not terminate.*

**Proof.** Assume for a contradiction that $(e_0, h_0, f_0, k_0)$ is well-typed in the memory-type system but it causes a memory error. Then we can prove that a faulty state can be well-typed, which conflicts with Proposition 2. Suppose a transition from $(e_0, h_0, f_0, k_0)$ to a faulty state $(e_n, h_n, f_n, k_n)$:

$$(e, h, f, k) \leadsto (e_1, h_1, f_1, k_1) \leadsto \cdots \leadsto (e_n, h_n, f_n, k_n).$$

We can prove that every $(e_i, h_i, f_i, k_i)$ is well-typed by induction on $i$.

- case $i = 0$: The assumption is that $\vdash (e_0, h_0, f_0, k_0)$.
- case $i > 0$: By induction hypothesis, $\vdash (e_{i-1}, h_{i-1}, f_{i-1}, k_{i-1})$. Since there exists a transition $(e_{i-1}, h_{i-1}, f_{i-1}, k_{i-1}) \leadsto (e_i, h_i, f_i, k_i)$, by Proposition 1, $\vdash (e_i, h_i, f_i, k_i)$.

Therefore a well-typed state does not go to a stuck state. □

### 5.3. Transformed programs are well-typed

Now we prove that programs transformed by our algorithm do not cause any memory error. There are two key propositions.

- Transformed expressions respect preservation constraints: our algorithm does not insert any memory-free command that violates preservation constraints (Proposition 3).
- Transformed expressions are well-typed: for each transformed expression, there is a corresponding judgment in the memory-type system which is based on the result of our analysis and transformation (Proposition 4).

In order to achieve the above two key propositions, we first prove for two sub-routines of the algorithm.

- One is freeCond in Fig. 5 which takes a bound $\mathcal{B}$, a preservation constraint $\mathcal{E}$, and a multiset formula $L$, and gives a safe condition for deallocating the heap cells in $L$ without violating preservation constraint $\mathcal{E}$ under bound $\mathcal{B}$ (Lemma 5).
- The other is reduce which takes a bound $\mathcal{B}$ and a multiset formula $L$ and gives a multiset formula which is greater than or equal to $L$ under bound $\mathcal{B}$ (Lemma 6).

**Lemma 5.** *For a bound $\mathcal{B}$, a preservation constraint $\mathcal{E}$, and multiset formulas $L$, $L_1$, and $L_2$, when $\mathcal{C}_{\text{ns}} = (\beta_{\text{ns}} \Rightarrow \text{SET}(A))$, the following are true:*

(1) $(\mathcal{B} \wedge \mathcal{C}_{\text{ns}}) \Rightarrow (\mathsf{noSharing}_{\mathcal{B}}(L) \Rightarrow \text{SET}(L))$;
(2) $(\mathcal{B} \wedge \mathcal{C}_{\text{ns}}) \Rightarrow (\mathsf{disjoint}_{\mathcal{B}}(L_1, L_2) \Rightarrow L_1 \# L_2)$; *and*
(3) $(\mathcal{B} \wedge \mathcal{C}_{\text{ns}}) \Rightarrow \left(\left\{\mathsf{freeCond}_{\mathcal{B},\mathcal{E}}(L) \hookrightarrow L\right\} \# \mathcal{E}\right)$.

**Proof.** The proof is in [11]. □

**Lemma 6.** *For a bound $\mathcal{B}$ and a multiset formula $L$, $\mathsf{reduce}_{\mathcal{B}}(L)$ gives a multiset formula $L_R$ in a reduced form such that $\mathcal{B} \Rightarrow L \sqsubseteq L_R$.*

**Proof.** The proof is in [11]. □

**Proposition 3** (*Transformed Expressions Respect Constraints*). *For a bound $\mathcal{B}$, a preservation constraint $\mathcal{E}$, a boolean value $b$, and an expression $e$, if $e$ is transformed to $e'$ by the algorithm (i.e., $\mathcal{B}, \mathcal{E}, b \rhd e^{(\Delta,\mathcal{B}',\mu,L)} \Rightarrow e' : \mathcal{E}'$), then $(\mathcal{B} \wedge \mathcal{C}_{\text{ns}}) \Rightarrow \mathcal{E}' \# \mathcal{E}$ holds where $\mathcal{C}_{\text{ns}} = \beta_{\text{ns}} \Rightarrow \text{SET}(A)$.*

**Proof.** We prove it by induction on the number of calls:

- **case** (I-VALUE and I-NOF): $\mathcal{E}' = \emptyset$.
- **case** (I-FREE): Since $b' = \mathsf{freeCond}_{\mathcal{B},\mathcal{E}''}(L)$ where $\mathcal{E}'' = \mathcal{E} \cup \{b \hookrightarrow \mathsf{collapse}(\mu)\}$, by Lemma 5, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \left\{b' \hookrightarrow L\right\} \# \mathcal{E}''$. Therefore $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \left\{b' \hookrightarrow L\right\} \# \mathcal{E}$ also holds.
- **case** (I-CASE): By induction hypothesis, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{E}_i \# \mathcal{E}$ for $i = 1$ or 2. Then by definition, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow (\mathcal{E}_1 \cup \mathcal{E}_2) \# \mathcal{E}$ also holds.
- **case** (I-LET): By induction, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{E}_1 \# (\mathcal{E} \cup \{\text{true} \hookrightarrow L, b \hookrightarrow \mathsf{collapse}(\mu)\})$ and $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{E}_2 \# (\mathcal{E} \cup \mathcal{E}_1)$; that is, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{E}_i \# \mathcal{E}$ for $i = 1$ or 2. Then by definition, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow (\mathcal{E}_1 \cup \mathcal{E}_2) \# \mathcal{E}$ holds.
- **case** (I-APP): By Lemma 5, $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \left\{b' \hookrightarrow L \dot{\setminus} R\right\} \# \mathcal{E}$. □

Our analysis and transformation system always gives well-typed programs in our memory-type system. That is, for each transformed expression, there is a corresponding judgment in the memory-type system which is based on the result of our analysis and transformation.

**Proposition 4** (*Transformed Expressions are Well-Typed*). *The following are true:*

(1) *For a value $v$, if the algorithm transform $v$ to $v'$ (i.e., $\rhd v^{(\Delta,\mu)} \Rightarrow v'$), then $\Delta \vdash v : \mu \,\&\, \text{true}$ holds.*

(2) *For a bound $\mathcal{B}_0$, a preservation constraint $\mathcal{E}_0$, a boolean value $b$, and an expression $e$, if the algorithm transform $e$ to $e'$ (i.e., $\mathcal{B}_0, \mathcal{E}_0, b \rhd e^{(\Delta,\mathcal{B},\mu,L)} \Rightarrow e' : \mathcal{E}$), when $\mathcal{V}$ is a set of fresh names introduced during the analysis phase (i.e., $\Delta \rhd e : \mathcal{B}, \mu, L$),*

(a) *when* $b = \text{false}$, *there exists $\mathcal{C}$ such that $(\mathcal{B}_0 \wedge \mathcal{C}_{\text{ns}}) \Rightarrow \mathcal{C}$ and*

$$\mathcal{T}(\Delta) \vdash e' : \exists\mathcal{V}.\,(\mathcal{B}, \mathcal{T}(\mu), L, \mathcal{E})\,\&\,\mathcal{C};\ \textit{and}$$

(b) *when* $b = \text{true}$, *there exists fresh $R$ and $\mathcal{C}$ such that $(\mathcal{B}_0 \wedge \mathcal{C}_{\text{ns}}) \Rightarrow \mathcal{C}$ and*

$$\mathcal{T}(\Delta) \vdash e' : \exists\mathcal{V}.\,\left(\mathcal{B} \cup \{R \mapsto \mathsf{collapse}(\mu)\}, R, L, \mathcal{E}'\right)\&\,\mathcal{C}$$

$$\textit{where}\quad \mathcal{E}' = (\mathcal{E}\backslash R) \cup \left\{\text{true} \hookrightarrow (\dot\sqcup_{X\in\mathcal{V}} X)\dot\backslash R\right\}$$

$$\textit{and}\quad \mathcal{E}\backslash R \overset{\Delta}{=} \left\{b \hookrightarrow L\dot\backslash R \mid (b \hookrightarrow L) \in \mathcal{E}\right\}.$$

**Proof.** In proof, we do not explicitly put the translation function $\mathcal{T}$ because it is clear from the context where $\mathcal{T}$ should appear.

- **case** (I-FUN/U-FUN): The assumption is that $\rhd(\texttt{fix}\ y\ \lambda x.e)^{(\Delta,\mu)} \Rightarrow (\texttt{fix}\ y\ \lambda x.e')$ is derived by (I-FUN) and the last step of (U-FUN); that is,

$$\mu = \forall A.A \to \exists X.(L_1, L_2)\ \text{and} \tag{44}$$

$$\mathcal{B}, \{\neg\beta \hookrightarrow A\}, \text{true} \rhd e^{(\{f\mapsto\mu, x\mapsto A\},\mathcal{B},\mu',L)} \Rightarrow e' : \mathcal{E} \tag{45}$$

where $L' = \mathsf{collapse}(\mu')$, $L_1 = \mathcal{S}(\mathsf{reduce}_{\mathcal{B}}(L'))$, $L_2 = \mathcal{S}(\mathsf{reduce}_{\mathcal{B}}(L))$, $\mathcal{S} = \{X/X_1, \ldots, X/X_n\}$, and $X_i$s are new $X$s in $\mathcal{V}$. By induction hypothesis, (45) implies that there exists $\mathcal{C}$ such that

$$\{f \mapsto \mu, x \mapsto A\} \vdash e' : \exists\mathcal{V} \cup \{R\}\,.$$
$$\left(\mathcal{B} \cup \left\{R \mapsto L'\right\}, R, L, (\mathcal{E}\backslash R) \cup \left\{\text{true} \hookrightarrow (\dot\sqcup_i X_i)\dot\backslash R\right\}\right)\&\mathcal{C} \tag{46}$$

$$\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{C}. \tag{47}$$

By Lemma 6,

$$\mathcal{B} \Rightarrow (L' \sqsubseteq \mathsf{reduce}_{\mathcal{B}}(L')) \wedge (L \sqsubseteq \mathsf{reduce}_{\mathcal{B}}(L)). \tag{48}$$

Note that these reduced forms consist of only $A$ and $X_i$s in $\mathcal{V}$. For a reduced form $L$, when $\mathcal{S}' = \left\{(\dot\sqcup_i X_i)/X\right\}$, we have $L \sqsubseteq \mathcal{S}'(\mathcal{S}L)$ because $\mathcal{S}'\mathcal{S} = \left\{(\dot\sqcup_i X_i)/X_1, \ldots, (\dot\sqcup_i X_i)/X_n\right\}$. Then (48) implies that

$$\mathcal{B} \Rightarrow (L' \sqsubseteq \mathcal{S}'L_1) \wedge (L \sqsubseteq \mathcal{S}'L_2). \tag{49}$$

By Proposition 3, (45) implies that $\mathcal{B} \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{E}\#\{\neg\beta \hookrightarrow A\}$, and

$$\mathcal{E}\#\{\neg\beta \hookrightarrow A\} \Rightarrow \mathcal{E} \sqsubseteq (\mathcal{E}\backslash A) \cup \{\beta \hookrightarrow A\}$$

because
· when $\beta = \text{false}$, $\mathcal{E}\#\{A\} \Rightarrow \mathcal{E} = \mathcal{E}\backslash A$, and
· when $\beta = \text{true}$, $\text{true} \Rightarrow \mathcal{E} \sqsubseteq (\mathcal{E}\backslash A) \cup \{\text{true} \hookrightarrow A\}$.
Then

$$\mathcal{E}\backslash R \sqsubseteq ((\mathcal{E}\backslash A)\backslash R) \cup \left\{\beta \hookrightarrow A\dot\backslash R\right\}. \tag{50}$$

Moreover, $\mathcal{E} \sqsubseteq \left\{\text{true} \hookrightarrow\ \dot\sqcup\,\text{free}(\mathcal{E})\right\}$ and by Lemma 6,

$$\mathcal{B} \Rightarrow\ \dot\sqcup\,\text{free}(\mathcal{E}) \sqsubseteq_{\mathsf{set}} \mathsf{reduce}_{\mathcal{B}}(\dot\sqcup\,\text{free}(\mathcal{E})).$$

Since the reduced form consists of $A$ or new $X_i$s in $\mathcal{V}$,

$$\mathsf{reduce}_{\mathcal{B}}(\dot{\sqcup}\,\mathrm{free}(\mathcal{E})) \sqsubseteq_{\mathsf{set}} A \mathbin{\dot{\sqcup}} (\dot{\sqcup}_{X_i \in \mathcal{V}} X_i).$$

Then (50) implies that

$$\begin{aligned}\mathcal{B} \Rightarrow \mathcal{E}\backslash R \;&\sqsubseteq\; \left\{\mathrm{true} \hookrightarrow ((A \mathbin{\dot{\sqcup}} (\dot{\sqcup}_{X_i\in\mathcal{V}} X_i))\dot{\backslash} A)\dot{\backslash} R,\; \beta \hookrightarrow A\dot{\backslash} R\right\}\\ &=\; \mathcal{S}'\left\{\mathrm{true} \hookrightarrow X\dot{\backslash} R, \beta \hookrightarrow A\dot{\backslash} R\right\}\end{aligned} \tag{51}$$

because $A \# X_i$. Then by (WEAK), (46), (47), (49), and (51) imply that

$$\begin{aligned}&\{f \mapsto \mu, x \mapsto A\} \vdash e' : \exists\mathcal{V} \cup \{R\}.\\ &\quad \left(\mathcal{B} \cup \left\{R \mapsto \mathcal{S}'L_1\right\}, R, \mathcal{S}'L_2, \mathcal{S}'\left\{\mathrm{true} \hookrightarrow X\dot{\backslash} R, \beta \hookrightarrow A\dot{\backslash} R\right\}\right) \& \mathcal{C}_{\mathrm{ns}}.\end{aligned}$$

By (MERGE),

$$\begin{aligned}&\{f \mapsto \mu, x \mapsto A\} \vdash e' : \exists\mathrm{dom}(\mathcal{B}) \cup \{X, R\}.\\ &\quad \left(\mathcal{B} \cup \{R \sqsubseteq L_1\}, R, L_2, \left\{\mathrm{true} \hookrightarrow X\dot{\backslash} R, \beta \hookrightarrow A\dot{\backslash} R\right\}\right) \& \mathcal{C}_{\mathrm{ns}}.\end{aligned}$$

Since the result part has only free names $A$, $X$, and $R$, by (WEAK),

$$\begin{aligned}&\{f \mapsto \mu, x \mapsto A\} \vdash e' :\\ &\quad \exists\{X, R\}. \left(\{R \mapsto L_1\}, R, L_2, \left\{\mathrm{true} \hookrightarrow X\dot{\backslash} R, \beta \hookrightarrow A\dot{\backslash} R\right\}\right) \& \mathcal{C}_{\mathrm{ns}}.\end{aligned}$$

By (FUN) and the definition of $\mathcal{T}$ in Section 5.1, $\Delta \vdash \mathtt{fix}\ f\ \lambda x.e' : \mathcal{T}(\mu)$.

- **case** (I-FREE/U-NODE): The assumption is that when $e = \mathtt{free}\ x\ \mathtt{when}\ b'; \mathtt{Node}(v_1', v_2')$ which is $\mathtt{let}\ y = \mathtt{free}\ x\ \mathtt{when}\ b'\ \mathtt{in}\ \mathtt{Node}(v_1', v_2')$ for some fresh $y$,

$$\mathcal{B}_0, \mathcal{E}_0, b \rhd \mathtt{Node}(v_1, v_2)^{(\Delta,\emptyset,\mu,\emptyset)} \Rightarrow e : \left\{b' \hookrightarrow L\right\}$$

where $\mu = \langle X, \mu_1, \mu_2\rangle$ is derived by (I-FREE) and (U-NODE); that is,

$$b' = \mathsf{freeCond}_{\mathcal{B}_0,\mathcal{E}_0'}(L), \tag{52}$$

$$\mathcal{E}_0' = \mathcal{E}_0 \cup \{b \hookrightarrow \mathsf{collapse}(\mu)\}, \tag{53}$$

$$\Delta(x) = \left\langle L, \mu_1', \mu_2'\right\rangle \text{ for some } \mu_1' \text{ and } \mu_2', \text{ and} \tag{54}$$

$$\rhd v_i^{(\Delta,\mu_i)} \Rightarrow v_i'. \tag{55}$$

By induction hypothesis, (55) implies that $\Delta \vdash v_i' : \mu_i \,\&\, \mathrm{true}$. By (NODE),

$$\Delta \vdash \mathtt{Node}(v_1', v_2') : \exists\{X\}. (\emptyset, \mu, \emptyset, \emptyset) \,\&\, \mathrm{true}.$$

Since $y$ is fresh, by Lemma 4,

$$\Delta \cup \{y \mapsto \emptyset\} \vdash \mathtt{Node}(v_1', v_2') : \exists\{X\}. (\emptyset, \mu, \emptyset, \emptyset) \,\&\, \mathrm{true}. \tag{56}$$

Since $\Delta(x) = \left\langle L, \mu_1', \mu_2'\right\rangle$, by (ID), $\Delta \vdash x : \left\langle L, \mu_1', \mu_2'\right\rangle \&\, \mathrm{true}$. By (FREE),

$$\Delta \vdash \mathtt{free}\ x\ \mathtt{when}\ b' : \exists\emptyset. (\emptyset, \emptyset, \emptyset, \left\{b' \hookrightarrow L\right\}) \,\&\, \mathrm{true}. \tag{57}$$

By (LET), (56) and (57) imply that

$$\Delta \vdash e : \exists\{X\}. (\emptyset, \mu, \emptyset, \left\{b' \hookrightarrow L\right\}) \,\&\, \mathrm{true} \tag{58}$$

which proves for $b = \mathrm{false}$.

Now we prove for $b = \text{true}$ with $\mathcal{C} = (b' \Rightarrow L\#\mathsf{collapse}(\mu))$. Since $\mathcal{C} \Rightarrow \{b' \hookrightarrow L\} \sqsubseteq_{\mathsf{set}} \{b' \hookrightarrow L \dot{\setminus} \mathsf{collapse}(\mu)\}$ and $\mu \sqsubseteq \mathsf{collapse}(\mu)$, by (WEAK), (58) implies that

$$\Delta \vdash e : \exists\{X\}.\left(\emptyset, \mathsf{collapse}(\mu), \emptyset, \left\{b' \hookrightarrow L \dot{\setminus} \mathsf{collapse}(\mu)\right\}\right) \& \mathcal{C}.$$

By (RINT),

$$\Delta \vdash e : \exists\{X, R\}.\left(\{R \mapsto \mathsf{collapse}(\mu)\}, R, \emptyset, \left\{b' \hookrightarrow L \dot{\setminus} R\right\}\right) \& \mathcal{C}.$$

By Lemma 5, (52) implies that $\mathcal{B}_0 \wedge \mathcal{C}_{\text{ns}} \Rightarrow \{b' \hookrightarrow L\}\#\mathcal{E}'_0$. Since $\mathcal{E}'_0$ includes $(b \hookrightarrow \mathsf{collapse}(\mu))$ and $b = \text{true}$, $\mathcal{B}_0 \wedge \mathcal{C}_{\text{ns}} \Rightarrow \mathcal{C}$.

The proofs of other cases are in [11]. □

**Theorem 2** (*Algorithm 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 ($\mathcal{B}, \emptyset, \text{false} \rhd e^{(\emptyset,\mathcal{B},\mu,L)} \Rightarrow e' : \mathcal{E}$), then expression $e'$ does not cause a memory error.*

**Proof.** By Proposition 4, $\emptyset \vdash e' : \exists\mathcal{V}.(\mathcal{B}, \mu, L, \mathcal{E}) \& \mathcal{C}_{\text{ns}}$ for some $\mathcal{V}$. By Lemma 2, we can apply substitution $\mathcal{S} = \{\emptyset/A\}$ to the judgment. As a result,

$$\emptyset \vdash e' : \exists\mathcal{V}.(\mathcal{S}\mathcal{B}, \mathcal{S}\mu, \mathcal{S}L, \mathcal{S}\mathcal{E}) \& \text{true}.$$

By (HEAP), $\vdash \emptyset : \emptyset$. By (FREED), $\emptyset \vdash \emptyset : \emptyset$. By (NIL), $\{\bullet \mapsto \mu\} \vdash \epsilon : \exists\emptyset.(\emptyset, \emptyset, \emptyset, \emptyset) \& \text{true}$. Therefore by (STATE), $\vdash (e', \emptyset, \emptyset, \epsilon)$. Then by Theorem 1, $(e', \emptyset, \emptyset, \epsilon)$ does not go to a stuck state. □

## 6. Experiments

We experimented with the insertion algorithm with ML benchmark programs which use various data types such as lists, trees, and abstract syntax trees:

| program | lines | description |
|---|---|---|
| `sieve` | 18 | prime number computation (size=10000) |
| `qsort` | 24 | quick sort (size=10000) |
| `merge` | 30 | merging two ordered integer lists (size=10000) |
| `msort` | 61 | merge sort (size=10000) |
| `queens` | 66 | solving eight queen problem |
| `mirage` | 141 | an interpreter for a tiny non-deterministic programming language |
| `life` | 169 | “life” from the SML/NJ [19] benchmark suite (loop=50) |
| `kb` | 557 | “knuth-bendix” from the SML/NJ [19] benchmark suite |
| `k-eval` | 645 | an interpreter for a tiny imperative programming language |
| `nucleic` | 3230 | “nucleic” from the SML/NJ [19] benchmark suite |

We first pre-processed benchmark programs to monomorphic and closure-converted [13] programs, and then applied the algorithm to the pre-processed programs.

| program | lines | (1) total[a] | (2) reuse[a] | (2)/(1) | cost(s[b]) |
|---|---|---|---|---|---|
| `sieve` | 18 | 161112 | 131040 | 81.3% | 0.004 |
| `qsort` | 24 | 675925 | 617412 | 91.3% | 0.007 |
| `merge` | 30 | 120012 | 59997 | 50.0% | 0.007 |
| `msort` | 61 | 440433 | 390429 | 88.7% | 0.019 |
| `queens` | 66 | 118224 | 6168 | 5.2% | 0.017 |
| `mirage` | 141 | 208914 | 176214 | 84.4% | 0.114 |
| `life` | 169 | 84483 | 8961 | 10.6% | 0.113 |
| `kb` | 557 | 2747397 | 235596 | 8.6% | 0.850 |
| `k-eval` | 645 | 271591 | 161607 | 59.5% | 1.564 |
| `nucleic` | 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 system is compiled by the Objective Caml 3.04 native compiler [12], and executed in Sun Sparc 400 MHz, Solaris 2.7

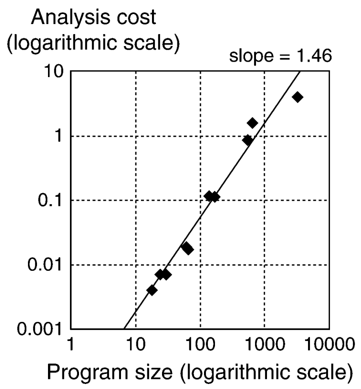

Fig. 14. Analysis cost and reuse ratio.

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 $\beta$ also keep the alias information of function parameters so that if two parameters share some heap cells, both of their dynamic flags $\beta$ 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 $\forall A.A \rightarrow \exists X.(L, L)$ where $L = (A \dot{\oplus} A) \dot{\sqcup} (X \dot{\oplus} X)$. (4) When we have multiple candidate cells for deallocation, we choose one whose guard is weaker than the others. For incomparable guards, we choose one arbitrarily.

The experimental results are shown in Fig. 14. Our analysis and transformation system achieves the memory reuse ratio (the fifth column) of 5.2% to 91.3%. In the table of Fig. 14, the second column is the number of lines, the third column is the amount of heap cells allocated during the execution of the original programs, the fourth the amount of heap cells reused during the execution of the transformed programs, the fifth its ratio, and the sixth the cost of our analysis and transformation. For the two cases whose reuse ratio is low (`queens` and `kb`), we found that they have a number of data structures that are shared. 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 for such cases, our experimental results are encouraging in terms of accuracy and cost. The graph in Fig. 14 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.

| program | reuse ratio | $(A)$ memory peak[a] | $(B)$ reduced peak | $(A-B)/A$ |
|---|---|---|---|---|
| `sieve` (size=1000) | 56.0% | 690 | 300 | 56.5% |
| `qsort` (size=100) | 81.0% | 1189 | 334 | 71.9% |
| `merge` (size=500) | 49.4% | 1197 | 606 | 49.4% |
| `msort` (size=100) | 82.5% | 714 | 321 | 55.0% |
| `queens` (n=5) | 8.3% | 255 | 255 | 0.0% |
| `mirage` | 84.4% | 1398 | 1361 | 2.6% |
| `life` (loop=5) | 10.6% | 2346 | 1746 | 25.6% |
| `kb` (group rule) | 12.7% | 27125 | 26501 | 2.3% |
| `k-eval` | 59.5% | 1044 | 944 | 9.6% |
| `nucleic` | 18.2% | 103677 | 89352 | 13.8% |

[a] words: the maximum number of live cells. It is profiled by our interpreter which has the same memory layout as that of Objective Caml 3.04 compiler [12]. $(A)$ is for the original program and $(B)$ is for the program transformed by our algorithm.

Fig. 15. The memory peak is reduced.

Our transformation reduces the memory peak from 0.0% to 71.9% (Figs. 15–17). The memory peak is the maximum number of live cells during the program execution. In Fig. 15, the second column is the reuse ratio, the third is the memory peak of the original programs, the fourth the memory peak of the transformed programs, and the fifth how much the memory peak is reduced by our transformation. For `sieve`, `merge`, `qsort`, and `msort`, both reuse ratios and peak reductions are high. For `queens` and `kb`, both reuse ratios and peak reductions are low. But for `life` and `mirage`, reuse ratios and peak reductions do not match. For `mirage`, its reuse ratio is high (84.4%) whereas its peak reduction is low (2.6%). This is because, as seen in the graph (f) of Fig. 16, the transformed `mirage` fails to reduce several peaks in the second phase. For `life`, the situation is reversed. This is because, as seen in the graph (e) of Fig. 16, it always reuses only those cells that contribute to the memory peak.

## 7. Conclusion and future work

We have presented a static analysis and a source-level transformation system that add explicit memory reuse commands into the program text, and we have shown that they effectively find memory reuse points.

We are currently implementing the analysis and transformation system 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 reducing this overhead by doing a constant

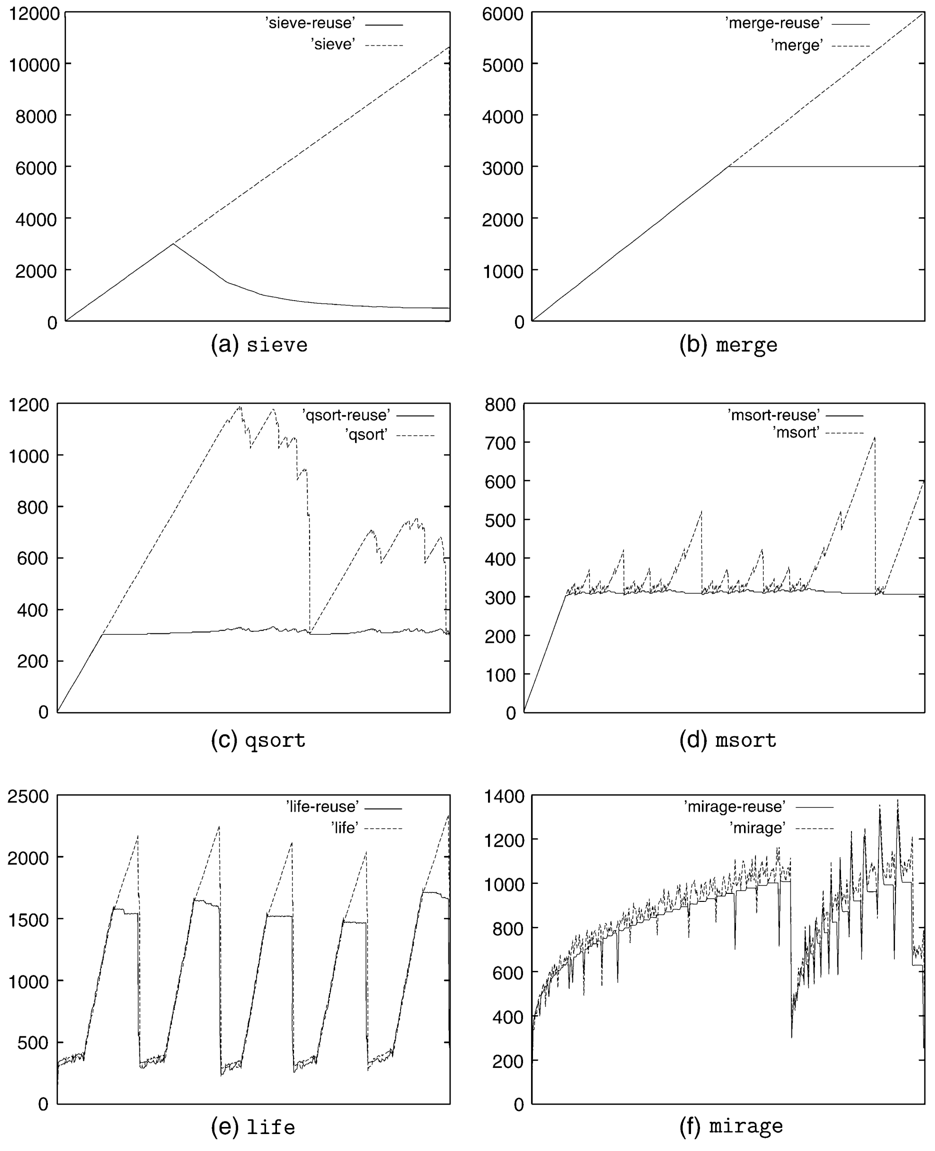

Fig. 16. The numbers of live memory cells from start to the end. The upper dotted lines are the original program's and the lower solid lines are those of the programs transformed by our algorithm.

propagation for dynamic flags; omitting some unnecessary flags; associating a single flag with several data constructors of the same size; implementing flags via 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

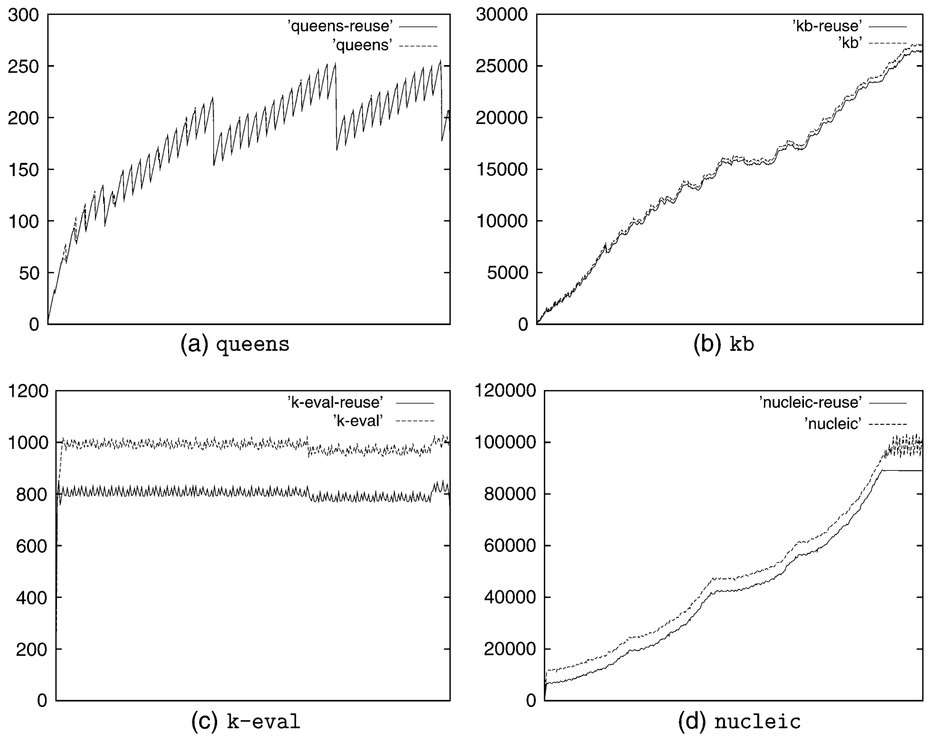

Fig. 17. The numbers of live memory cells from start to the end. The upper dotted lines are the original program's and the lower solid lines are those of the programs transformed by our algorithm.

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 $\alpha$ 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 $\beta$, $\beta_{\text{ns}}$, $\beta'$, and $\beta'_{\text{ns}}$, we pass $\beta \wedge \beta'$ and $\beta_{\text{ns}} \wedge \beta'_{\text{ns}}$. 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.

## Acknowledgments

We thank anonymous referees and colleagues for comments and suggestions that helped to improve our paper.

## References

[1] D. Aspinall, M. Hofmann, Another type system for in-place update, in: Proceedings of the European Symposium on Programming, in: Lecture Notes in Computer Science, vol. 2305, 2002, pp. 36–52.

[2] E. Barendsen, S. Smetsers, Uniqueness typing for functional languages with graph rewriting semantics, Mathematical Structures in Computer Science 6 (1995) 579–612.
[3] B. Blanchet, Escape analysis: Correctness proof, implementation and experimental results, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 1998, pp. 25–37.
[4] K. Crary, D. Walker, G. Morrisett, Typed memory management in a calculus of capabilities, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 1999, pp. 262–275.
[5] D. Gay, A. Aiken, Language support for regions, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2001, pp. 70–80.
[6] O. Gheorghioiu, A. Sălcianu, M. Rinard, Interprocedural compatibility analysis for static object preallocation, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 2003, pp. 273–284.
[7] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in Cyclone, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002, pp. 282–293.
[8] W.L. Harrison III, The interprocedural analysis and automatic parallelization of scheme programs, Lisp and Symbolic Computation 2 (3–4) (1989) 179–396.
[9] S. Ishtiaq, P. O'Hearn, BI as an assertion language for mutable data structures, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 2001.
[10] N. Kobayashi, Quasi-linear types, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 1999, pp. 29–42.
[11] O. Lee, H. Yang, K. Yi, Correctness proof on an algorithm to insert memory reuse commands into ML-like programs, Technical Memorandum ROPAS-2003-19, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University. http://ropas.snu.ac.kr/memo, November 2003.
[12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, J. Vouillon, The Objective Caml system release 3.06, Institut National de Recherche en Informatique et en Automatique. http://caml.inria.fr, August 2002.
[13] Y. Minamide, G. Morrisett, R. Harper, Typed closure conversion, in: Proceedings of the ACM Symposium on Principles of Programming Languages, 1996, pp. 271–283.
[14] M. Mohnen, Efficient compile-time garbage collection for arbitrary data structures, in: Proceedings of Programming Languages: Implementations, Logics and Programs, in: Lecture Notes in Computer Science, vol. 982, Springer-Verlag, 1995, pp. 241–258.
[15] nML programming language system, version 0.92b, Research On Program Analysis System, Seoul National University. http://ropas.snu.ac.kr/n, April 2004.
[16] P. O'Hearn, J.C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: Proceedings of the Annual Conference of the European Association for Computer Science Logic, 2001, pp. 1–19.
[17] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proceedings of the IEEE Symposium on Logic in Computer Science, 2002.
[18] F. Smith, D. Walker, G. Morrisett, Alias types, in: Proceedings of the European Symposium on Programming, in: Lecture Notes in Computer Science, vol. 1782, 2000, pp. 366–382.
[19] The Standard ML of New Jersey, version 110.0.7, Bell Laboratories, Lucent Technologies. http://cm.bell-labs.com/cm/cs/what/smlnj, October 2000.
[20] M. Tofte, L. Birkedal, A region inference algorithm, ACM Transactions on Programming Languages and Systems 20 (4) (1998) 734–767.
[21] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T.H. Olesen, P. Sestoft, Programming with regions in the ML Kit (for version 4), IT University of Copenhagen. http://www.it-c.dk/research/mlkit, April 2002.
[22] M. Tofte, J.-P. Talpin, Region-based memory management, Information and Computation 132 (2) (1997) 109–176.
[23] M. Tofte, J.-P. 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, 1994, pp. 188–201.
[24] D.N. Turner, P. Wadler, C. Mossin, Once upon a type, in: Proceedings of the International Conference on Functional Programming and Computer Architecture, 1995, pp. 25–28.
[25] P. Wadler, Linear types can change the world!, in: M. Broy, C. Jones (Eds.), Programming Concepts and Methods, North-Holland, Sea of Galilee, Israel, 1990.

[26] D. Walker, G. Morrisett, Alias types for recursive data structures, in: Proceedings of the Workshop on Types in Compilation, in: Lecture Notes in Computer Science, vol. 2071, 2000, pp. 177–206.
[27] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.