To be presented in the SPACE 2004: the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management, Venice, Italy, January 2004.

Experiments on the Effectiveness of an Automatic Insertion of Safe Memory Reuses into ML-like Programs*

Oukseh Lee Kwangkeun Yi
Research On Program Analysis System
School of Computer Science & Engineering
Seoul National University
{oukseh; kwang}@ropas.snu.ac.kr

ABSTRACT

We present extensive experimental results on our static analysis and source-level transformation [12, 11] that adds explicit memory-reuse commands into ML program text.

Our analysis and transformation cost is negligible (1,582 to 29,000 lines per seconds) enough to be used in daily programming. The payoff is the reduction of memory peaks and the total garbage collection time. The transformed programs reuse 3.8% to 88.6% of total allocated memory cells, and the memory peak is reduced by 0.0% to 71.9%. When the memory peak reduction is large enough to overcome the costs of dynamic flags and the memory reuse in the generational garbage collection, it speeds up program's execution by up to 25.4%. Otherwise, our transformation can slowdown programs by up to 42.9%. The speedup is likely only when the portion of garbage collection time among the total execution time is more than about 50%.

1. OVERVIEW

After observing promising yet preliminary experiment numbers on our automatic insertion of explicit memory-reuse commands into ML-like programs [12, 11], we need to gather more extensive numbers on where, if any, the cost-effectiveness of our analysis and transformation shines most. Only after having identified such strength in practicality of our analysis and transformation we can integrate them into our nML compiler system [16] and guide the programmers on when and for what they should turn on the optimization.

This paper reports experiment numbers regarding the following questions. How much effect does our transformation have on the program's memory behavior in terms of memory peak and garbage collection performance? How much overhead does our transformation have on the program's execution time?

Before we summarize the numbers, let us briefly overview our analysis, and how it is different from other related works.

Our static analysis and a source-level transformation [12] adds explicit memory-reuse commands into program text so that the program should not blindly request memory when constructing data. The explicit memory-reuse is by inserting explicit memory-free commands right before data-construction expressions. Because the unit of both memory-free and allocation is an individual cell, such memory-free and allocation sequences can be implemented as memory reuses.

*This work is supported by the Brain Korea 21 in 2003.

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

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

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

The type systems [26, 25, 2] based on linear logic fail to achieve Example 1 case because variable `l` is used twice. Kobayashi [10], and Aspinall and Hofmann [1] overcome this shortcoming by using more fine-grained usage aspects, but their systems still reject Example 1 because variable `l` and `t` are aliased at line `(2)`–`(3)`. They cannot properly handle aliasing: for "`let x=y in` e" where `y` points to a list, this list cannot in general be reused at e in their systems. Moreover, Aspinall and Hofmann did not consider an automatic transformation for reuse. Kobayashi provides an automatic transformation, but he requires the memory system to bookkeep a reference counter for every heap cell.

Deductive systems like the separation logic [9, 17, 18] and the alias-type system [19, 27] are powerful enough to reason about shared mutable data structures, but they cannot be used for our goal; they are not automatic. They need the programmer's help about memory invariants for loops or recursive functions.

The region-based memory managements [23, 24, 4, 5, 7] use a fixed partitioning strategy for recursive data structures, which is either implied by the programmer's region declarations or hard-wired inside the region-inference engine [21, 22]. 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 the relation's targets and deallocation's granularity. His relation is between memory objects linked from program variables and their binding expression's results. Ours is between memory objects linked from any two program variables. His deallocation is at the end of a `let` or function body. Transformations like the one in Example 1 are impossible in his system. Harrison's [8] and Mohnen's [15] escape analyses have similar limitation: the deallocations is at the end of function body.

Our experimental results show that for small to large ML benchmark programs:

- Our analysis and transformation cost is small; 1,582 to 29,000 lines per seconds in Pentium4, 3Ghz.
- The programs reuse 3.8% to 88.6% of total allocated memory cells. The small-ratio cases are for programs that have too much sharings among memory cells. Other than those "torturing" cases, our experimental results are encouraging in terms of accuracy and cost.
- The memory peak is reduced by 0.0% to 71.9% which is co-related to the memory reuse ratio in most cases. A high memory reuse ratio usually means a large reduction of memory peak. However, if the memory reuses are restricted only to those that contribute to the memory peak, even a small reuse ratio can result in a substantial reduction of memory peak. Similarly, if the memory reuses are frequent but are restricted only to those that do not contribute to the memory peak, even a large reuse ratio can result in a negligible reduction of memory peak.
- The garbage collection time has 88.0% speedup to 1.6% slowdown. The slowdown is when the shift of the garbage collection points due to the memory reuses meets more live cells than the original case.
- The runtime has 25.4% speedup to 42.9% slowdown. The speedup is due to the reduction of garbage collection time and the slowdown is due to the cost to handle dynamic flags and the cost of memory reuses in the generational garbage collection.

We can observe three co-relations:

- If our transformation reuses much then it consistently results in much reduction in the total garbage collection time.
- If our transformation reuses much then it results in much reduction in memory peak. But, some exceptions are possible; if reuses focus on those not contributing to the memory peak they can result in small reduction in memory peak.
- If our transformation reuses much and the portion of the garbage collection time is big, then program's execution time reduction occurs.

Our analysis and transformation cost is negligible enough to be used in daily programming. The payoff is very likely in reduction in memory peaks and the total garbage collection time. Reduction in program's execution time is likely only when the portion of garbage collection time among the total execution time is more than about 50%.

2. ANALYSIS AND TRANSFORMATION

We first briefly explain our analysis and transformation that adds safe memory reuse commands into ML-like programs [12]. The features of our analysis and transformation are:

- Partitioning of heap cells is pivoted by two axes: one by structures (e.g. heads and tails for lists, roots and subtrees for trees, etc.) and the other by set exclusions (e.g. cells A excluding B). This double-axed partitioning is expressive enough to isolate proper reusable cells from others.
- Sharing information among heap cells is maintained, in order to find the disjointness properties between two partitions of heap cells. An analysis result consists of terms called "*multiset formula*." 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 not done by reanalyzing a function body multiple times.
- Dynamic flags are inserted to functions in order to condition their memory-free commands on their call sites. Dynamic flags are simple boolean expressions.

Section 2.1 intuitively presents the features of our method for an example program. Section 2.2 presents the key abstract domain (memory-types) for our analysis. Section 2.3 shows, for the same example as in Section 2.1, a more detailed explanation on how our analysis and transformation works. The exact definition of our analysis and transformation is in [12].

2.1 Exclusion Among Heap Cells and Dynamic Flags

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

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

those nodes of the input that are *not* parts of the output. A concrete example is the following `copyleft` function. Both of its input and output are trees. The output tree's nodes along its left-most path are separate copies from the input tree and the rest are shared with the input tree.

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

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

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

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

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

2.2 The Abstract Domain for Heap Objects

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

To simplify the presentation, we consider only binary trees for heap objects. 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`. We explain how we handle arbitrary algebraic data types in Section 3.

2.2.1 Multiset Formula

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

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

$$\begin{array}{rcl} L & ::= & A \mid R \mid X \mid \pi.\mathsf{root} \mid \pi.\mathsf{left} \mid \pi.\mathsf{right} \\ & \mid & \emptyset \mid L \dot{\sqcup} L \mid L \dot{\oplus} L \mid L \dot{\setminus} L \end{array}$$

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

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

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

2.2.2 Memory-Types

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

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

A memory-type μ_{tree} for a tree-typed value abstracts a set of heap objects. A heap object is a pair $\langle v, h \rangle$ of a value v and a heap h that contains all the reachable cells from v. Intuitively, it represents a tree reachable from v in h when v is a location; otherwise, it represents `Leaf`. A memory-type is either in a *structured* or *collapsed* form. A structured memory-type is a triple $\langle L, \mu_1, \mu_2 \rangle$, and its meaning (concretization) is a set of heap objects $\langle l, h \rangle$ such that L, μ_1, and μ_2 abstract the location l and the left and right subtrees of $\langle l, h \rangle$, respectively. A collapsed memory-type is more abstract than a structured one. It is simply a multiset formula L, and its meaning (concretization) is a set of heap objects $\langle v, h \rangle$ such that L abstracts every reachable location and its sharing in $\langle v, h \rangle$. The formal meaning of memory-types is in Figure 1.

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

2.3 The Insertion Algorithm

We explain our analysis and transformation using the `copyleft` example in Section 2.1:

SEMANTICS OF MULTISET FORMULAS

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

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

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

where

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

REQUIREMENTS ON GOOD ENVIRONMENTS

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

SEMANTICS OF MEMORY-TYPES FOR TREES

$$\begin{array}{rrcl} v \in & \mathsf{Values} & \triangleq & \{\mathtt{Leaf}\} \cup \mathsf{Locations} \\ h \in & \mathsf{Heaps} & \triangleq & \mathsf{Locations} \xrightarrow{\text{fin}} \{(v_1, v_2) \mid v_i \text{ is a value}\} \end{array}$$

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

$$\begin{array}{rcl} [\![\langle L, \mu_1, \mu_2\rangle]\!]_{\mathsf{tree}}\,\eta & \triangleq & \{\langle l, h\rangle \mid h(l) = (v_1, v_2) \ \wedge\ [\![L]\!]\eta\, l \sqsupseteq 1 \ \wedge\ \langle v_i, h\rangle \in [\![\mu_i]\!]_{\mathsf{tree}}\,\eta\} \\ [\![L]\!]_{\mathsf{tree}}\,\eta & \triangleq & \left\{\langle l, h\rangle \;\middle|\; \begin{array}{l} l \in \mathrm{dom}(h) \wedge \forall l'.\ \mathsf{let}\ n = \text{number of different paths from } l \text{ to } l' \text{ in } h \\ \qquad\qquad \mathsf{in}\ (n \geq 1 \Rightarrow [\![L]\!]\eta\, l' \sqsupseteq 1) \wedge (n \geq 2 \Rightarrow [\![L]\!]\eta\, l' = \infty) \end{array} \right\} \\ & & \cup\, \{\langle \mathtt{Leaf}, h\rangle \mid h \text{ is a heap}\} \end{array}$$

Figure 1: The Semantics of Multiset Formulas and Memory-Types for Trees.

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

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

2.3.1 Step One: The Memory-Usage Analysis

Our memory-usage analysis computes memory-types for all expressions in `copyleft`. In particular, it gives 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. We start from the least memory-type $\forall A.A \to \exists X.(\emptyset, \emptyset)$ for a function. Each iteration assumes that the recursive function itself has the memory-type obtained in the previous step, and the argument to the function has the (fixed) memory-type A. Under this assumption, we calculate the memory-type and the used cells for the function body. To guarantee the termination, the resulting memory-type and the used cells are approximated by "widening" after each iteration.

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

- **Line `(1)`**: The memory-type for `Leaf` is $\emptyset$, which says that the result tree is empty.
- **Line `(2)`**: The `Node`-branch is executed only when `t` is a non-empty tree. We exploit this fact to refine the memory-type A of `t`. We partition A into three parts: the root cell named $\pi.\mathsf{root}$, the left subtree named

π.left, and the right subtree named π.right, and record that their collection is A: $\pi.\mathsf{root} \dot\cup (\pi.\mathsf{left} \dot\oplus \pi.\mathsf{right}) = A$. Then `t1` and `t2` have π.left and π.right, respectively.

The next step is to compute a memory-type of the recursive call "`copyleft t1`." In the previous iteration's memory-type $\forall A.\, A \to \exists X.(A \dot\cup X, A)$ of `copyleft`, we instantiate A by the memory-type π.left of the argument `t1`, and X by the name X_2 for the newly allocated cells at line `(2)`. The instantiated memory-type $\pi.\mathsf{left} \to (\pi.\mathsf{left} \dot\cup X_2,\ \pi.\mathsf{left})$ says that when applied to the left subtree `t1` of `t`, the function returns a tree consisting of new cells or the cells already in the left subtree `t1`, but uses only the cells in the left subtree `t1`. So, the function call's result has the memory-type $\pi.\mathsf{left} \dot\cup X_2$, and uses the cells in π.left. However, we use name R for the result of the function call, and record that R is included in $\pi.\mathsf{left} \dot\cup X_2$.

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

After analyzing the branches separately, we join the results from the branches. The memory-type for the `Leaf`-branch is $\emptyset$, and the memory-type for the `Node`-branch is $\langle X_3, R, \pi.\mathsf{right}\rangle$. We join these two memory-types by first collapsing $\langle X_3, R, \pi.\mathsf{right}\rangle$ to get $X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right})$, and then joining the two collapsed memory-types $X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right})$ and $\emptyset$. Note that when combining X_3 and $R \dot\oplus \pi.\mathsf{right}$, we use $\dot\cup$ instead of $\dot\oplus$: it is because a root cell abstracted by X_3 cannot be in the left or right subtree. So, the function body has the memory-type $X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right})$.

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

The last step of each fixpoint iteration is widening: reducing all the multiset formulas into simpler yet more approximated ones. We widen the result memory-type $X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right})$ and the used cells $\pi.\mathsf{root} \dot\cup \pi.\mathsf{left}$ with the records $\mathcal{B}(R) = \pi.\mathsf{left} \dot\cup X_2$ and $\mathcal{B}(\pi) = A$ which means $R \sqsubseteq \pi.\mathsf{left} \dot\cup X_2$ and $\pi.\mathsf{root} \dot\cup (\pi.\mathsf{left} \oplus \pi.\mathsf{right}) \sqsubseteq A$.

$$
\begin{array}{lll}
& X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right}) & \\
\sqsubseteq & X_3 \dot\cup ((\pi.\mathsf{left} \dot\cup X_2) \dot\oplus \pi.\mathsf{right}) & (R \sqsubseteq \pi.\mathsf{left} \dot\cup X_2) \\
= & X_3 \dot\cup (\pi.\mathsf{left} \dot\oplus \pi.\mathsf{right}) \dot\cup (X_2 \dot\oplus \pi.\mathsf{right}) & \\
& & (\dot\oplus \text{ distributes over } \dot\cup) \\
\sqsubseteq & X_3 \dot\cup A \dot\cup (X_2 \dot\oplus \pi.\mathsf{right}) & (\pi.\mathsf{left} \dot\oplus \pi.\mathsf{right} \sqsubseteq A) \\
\sqsubseteq & X_3 \dot\cup A \dot\cup (X_2 \dot\oplus A) & (\pi.\mathsf{right} \sqsubseteq A) \\
= & X_3 \dot\cup A \dot\cup X_2 \dot\cup A & (A \text{ and } X_2 \text{ are disjoint})
\end{array}
$$

Finally, by replacing all the newly introduced X_i's by a fixed name X and by removing redundant A and X, we obtain $A \dot\cup X$. The used cells $\pi.\mathsf{root} \dot\cup \pi.\mathsf{left}$ is reduced to A because $\pi.\mathsf{root} \dot\cup \pi.\mathsf{right} \sqsubseteq A$

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.[1] The presence of $\dot\oplus$ and A in L indicates whether the result tree has a shared sub-part: if neither $\dot\oplus$ nor A is present in L, the result tree 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.[2]

2.3.2 Step Two: Free Commands Insertion

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

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

Our algorithm transforms the `copyleft` function as follows:

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

The algorithm decides to pass $\beta \wedge \beta_{\mathrm{ns}}$ and β_{ns} in the recursive call `(2)`. To find the first parameter, we collect constraints about conditions for which heap cells we should not free. Then, the candidate heap cells to deallocate must be disjoint with the cells to preserve. We derive such disjointness condition, expressed by a simple boolean expression. A preservation constraint has the conditional form $b \Rightarrow L$: when b holds, we should not free the cells in multiset L because, for instance, they have already been freed, or will be used later. For the first parameter, we get two constraints "$\neg\beta \Rightarrow A$" and "true $\Rightarrow X_3 \dot\cup (R \dot\oplus \pi.\mathsf{right})$." The first constraint means that we should not free the cells in the argument tree `t` if β is false, and the second that we should not free the cells in the result tree of the `copyleft` function. Now the candidate heap cells to deallocate inside the recursive call's body are $\pi.\mathsf{left} \dot\setminus R$ (the heap cells for `t1` excluding those in the result of the recursive call). For each constraint $b \Rightarrow L$, the algorithm finds a boolean expression which guarantees that L and $\pi.\mathsf{left} \dot\setminus R$ are disjoint if b is true; then, it takes the conjunction of all the found boolean expressions.

- For "$\neg\beta \Rightarrow A$," the algorithm concludes that A and $\pi.\mathsf{left} \dot\setminus R$ may overlap because $\pi.\mathsf{left} \sqsubseteq A$. Thus the algorithm takes "$\neg\beta \Rightarrow$ false," equivalently, β.
- For "true $\Rightarrow X_3 \dot\cup (R \oplus \pi.\mathsf{right})$," the algorithm finds out that flag β_{ns} ensures that $X_3 \dot\cup (R \oplus \pi.\mathsf{right})$ and $\pi.\mathsf{left} \dot\setminus R$ are disjoint:

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

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

– X_3 and $\pi.\mathsf{left}\dot{\backslash}R$ are disjoint because $\pi.\mathsf{left} \sqsubseteq A$, and X_3 and A are disjoint;

– R and $\pi.\mathsf{left}\dot{\backslash}R$ are disjoint because R is excluded in $\pi.\mathsf{left}\dot{\backslash}R$; and

– $\pi.\mathsf{right}$ and $\pi.\mathsf{left}\dot{\backslash}R$ are disjoint when β_{ns} is true because $\pi.\mathsf{right} \oplus \pi.\mathsf{left} \sqsubseteq A$ and β_{ns} ensures no sharing of argument A's sub-parts.

Thus the algorithm takes "true $\Rightarrow \beta_{\mathrm{ns}}$," equivalently, β_{ns}.

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

For the second boolean flag in the recursive call `(2)`, we find a boolean expression that ensures no sharing of a sub-part inside the left subtree `t1`. We use the memory-type $\pi.\mathsf{left}$ of `t1`, and find a boolean expression that guarantees no sharing inside the multiset $\pi.\mathsf{left}$; β_{ns} becomes such an expression because $\pi.\mathsf{left} \sqsubseteq A$ and β_{ns} ensures no sharing of argument A's sub-parts.

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

3. EXPERIMENT NUMBERS

We experimented our insertion algorithm with ML benchmark programs which use various data types such as lists, trees, and abstract syntax trees. We first pre-processed benchmark programs to monomorphic and closure-converted [14] programs, and then applied the algorithm to the pre-processed programs.

We extended the algorithm to treat programs with more features:

- 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.

- For functions with several parameters, we made the dynamic flag also keep the alias information between function parameters so that if two parameters share some heap cells, both of their dynamic flags β are turned off.

- For higher-order cases, we simply assumed the worst memory-types for the argument functions. For instance, we just assumed that an argument function, whose type is $\mathsf{tree} \to \mathsf{tree}$, has memory-type $\forall A.A \to \exists X.(L, L)$ where $L = (A \dot{\oplus} A) \dot{\sqcup} (X \dot{\oplus} X)$.

- When we have multiple candidate cells for deallocation, we chose one whose guard is weaker than the others. For incomparable guards, we arbitrarily chose one.

3.1 Analysis Cost, Memory Reuse Ratio, and Memory Peak

The cost of the analysis and transformation ranges from 1,582 to 29,000 lines per seconds in Pentium4 (column A in Figure 2.(a)). The graph (b) in Figure 2 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.

Our analysis and transformation achieves the memory reuse ratio of 3.8% to 88.6% (column C in Figure 2.(a)). For the two cases whose reuse ratio is low (`queens` and `kb`), we found that they have too much sharing. The `kb` program heavily uses a term-substitution function that can return a shared structure, where the number of shares depends on an argument value (e.g. a substitution item e/x has every x in the target term share e). Other than such cases, our experimental results are encouraging in terms of accuracy and cost.

Our transformation reduces the memory peak from 0.0% to 71.9% (column E in Figure 2.(a)). The memory peak is the maximum number of live cells during the program execution. 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 Figure 3, 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 (g) of Figure 3, it always reuses only those cells that contribute to the memory peak.

3.2 Garbage Collection and Execution Overhead

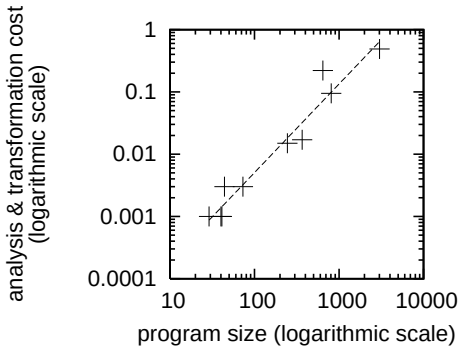
The GC portion of execution time of the benchmark programs ranges from 1.1% to 81.3% and our transformation reduces the GC time by −1.6% to 88.5%. The reduction of GC time by our transformation is co-related to memory reuse ratio: the reduction of total GC time is high when the reuse ratio is high. See column C in Figure 4.(a) and (b), and graph (c) in Figure 4.

The runtime overhead of the added dynamic flags are not much; it ranges from −2.4% to 8.3% of the program's total execution time. See column D of Figure 4.(a) and (b). We isolated the overhead by measuring the execution time of transformed program without executing the memory reuses. We implemented a set of dynamic flags as a bit-vector but we did not do any other optimization such as a constant propagation on dynamic flags. We, at the moment, have no explanation about the unexpected case that the dynamic flags actually improve the execution time up to 2.4% (`k-eval`, column D of Figure 4.(b)). We suspect some pressure on the entangled code generation for the Sun's UltraSparc has been released by the extra flag parameters to functions.

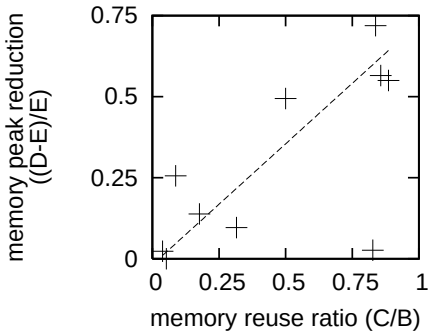
Our transformation has a mixed influence on the program's total execution time. See column E of Figure 4.(a) and (b), and the graph (d) in Figure 4. Our transformation can improve the program's execution time by up to 25.4%, and can also slowdown the execution by up to 42.9%. For `qsort` whose both reuse ratio and the GC portion in the total execution time are high, our transformation shortens the runtime by 25.4%. For `mirage` whose reuse ratio is high but

program	lines	A analysis cost	B allocation	C reuse	C/B	D memory peak	E memory peak (reuse)	(D-E)/E
`sieve`[a]	29	0.001	30829	26423	85.7%	690	300	56.5%
`merge`[b]	40	0.001	5860	2930	50.0%	1197	606	49.4%
`qsort`[c]	41	0.001	35997	30148	83.7%	1189	334	71.9%
`queens`[d]	44	0.003	34641	1807	5.2%	255	255	0.0%
`msort`[c]	73	0.003	21506	19064	88.6%	714	321	55.0%
`mirage`[e]	245	0.015	20381	16857	84.4%	1398	1361	2.6%
`life`[f]	366	0.017	10036	875	8.7%	2346	1746	25.6%
`k-eval`[g]	645	0.220	52894	16684	31.5%	1044	944	9.6%
`kb`[f]	808	0.095	24473	940	3.8%	27125	26501	2.3%
`nucleic`[f]	3019	0.488	31092	5491	17.7%	103677	89352	13.8%

(a) Analysis cost, memory reuse ratio, and memory peak reduction



(b) Analysis complexity (slope=1.46)



(c) Memory peak reduction is co-related to memory reuse ratio with some exceptions.

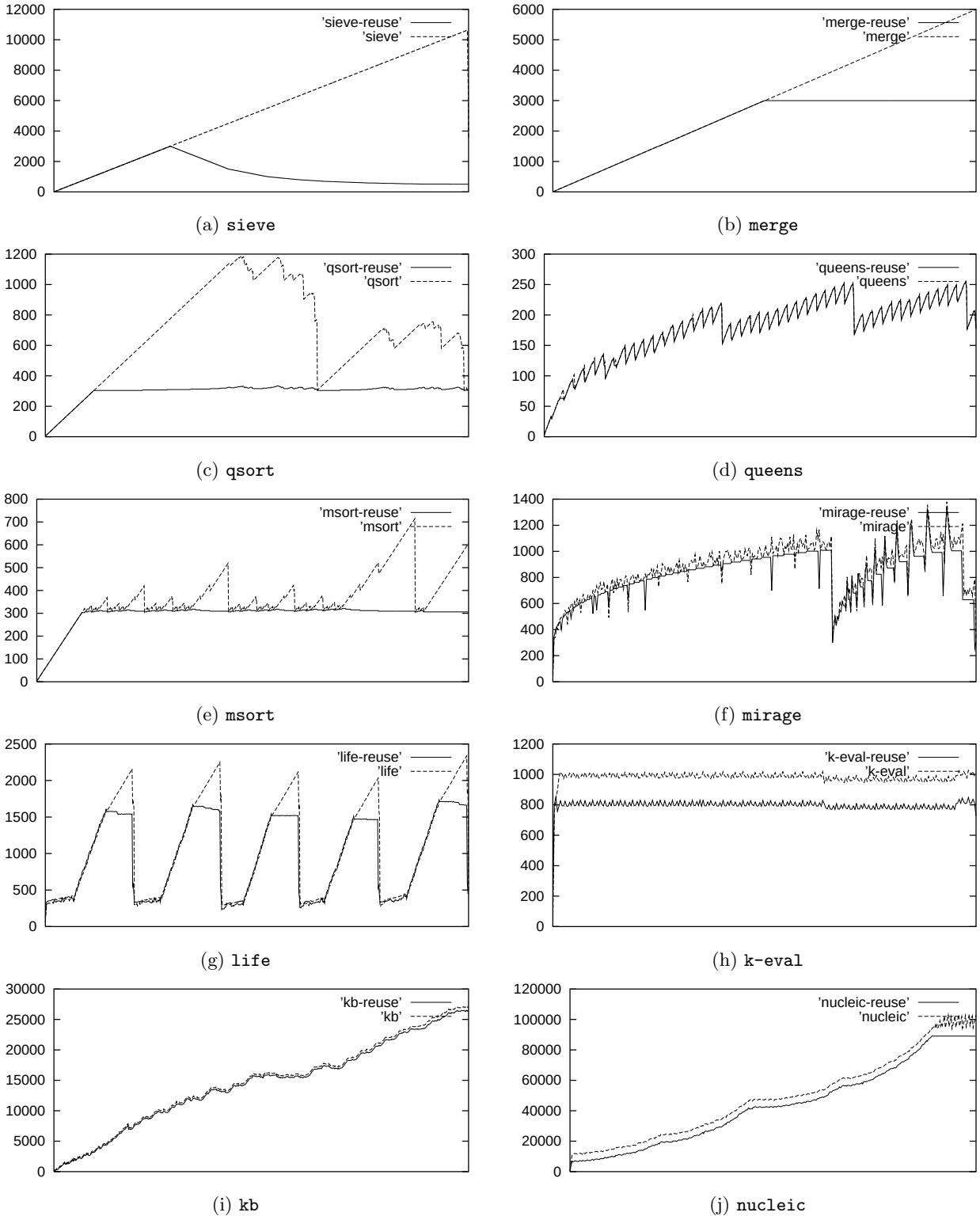
A seconds: the cost of our analysis and transformation, which is compiled by the Objective Caml 3.06 native compiler [13] and executed in Intel Pentium4 3GHz, Linux RedHat 9.0.
B kilo words: the amount of total allocated heap cells during the execution of original programs.
C kilo words: the amount of reused heap cells by our transformation.
D words: the maximum number of live cells during the execution of the original programs. It is profiled by our interpreter which has the same memory layout as that of Objective Caml 3.06 compiler [13]. In order to profile it in our interpreter, we reduce the parameters of benchmark programs.
E words: the maximum number of live cells during the execution of the programs transformed by our algorithm.
a prime number computation by the sieve of Eratosthenes (size = 30000, iteration=50)
b merging two ordered integer lists to an ordered list (size = 10000, iteration=50)
c quick/merge sort of an integer list (size=10000, iteration=50)
d eight queen problem (iteration=300)
e an interpreter for a tiny non-deterministic programming language (iteration=100)
f the benchmark programs from Standard ML of New Jersey [20] benchmark suite: `life` (generation=5, iteration=100), `kb` (iteration=5), and `nucleic`.
g an interpreter for a tiny imperative programming language (iteration=200)

Figure 2: Experimental results.

GC portion is almost nothing, our transformation increases its runtime by 42.9%.

The memory reuse speeds up five benchmark programs by reducing the GC time. `merge`, `qsort`, and `msort` become fast (74.6%–96.6%) because our transformation reduces much (38.4%–88.5%) of their big GC portion (50.7%–81.3%). The cases of `nucleic` and `sieve` are similar but they become slightly fast or slow due to the overhead of dynamic flags and memory reuses (98.4%–114.1%).

In our implementation, a memory reuse is more expensive than an allocation. We modify Objective Caml compiler [13] to initialize every new memory cell as mutable cell, and to translate a memory reuse command into the mutable-cell-update operator (`<-`) in Objective Caml. In the Objective Caml compiler, a mutable-cell-update takes more cost than an allocation does. It is compiled into codes for a function call that does many bookkeepings. Since Objective Caml employs a two generational garbage collection which maintains a reference table to keep pointers from the old generation to the young generation, in order to store a pointer to the young generation into a cell in the old generation, we have to update the reference table. Moreover, since the old generation heap of Objective Caml has an incremental mark-and-sweep garbage collector, in order to remove a pointer to a cell in the old generation, we have to modify the cell's marking tag. Although our current implementation is



(a) `sieve` (b) `merge`

(c) `qsort` (d) `queens`

(e) `msort` (f) `mirage`

(g) `life` (h) `k-eval`

(i) `kb` (j) `nucleic`

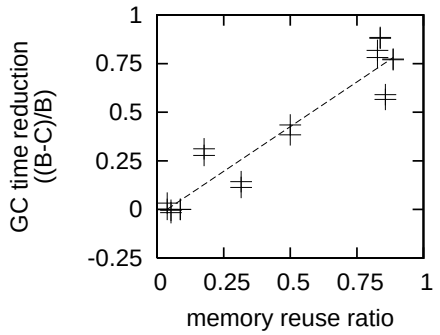
Figure 3: The memory peak behavior: the numbers of live memory cells from start to the end. The upper dotted lines are the original program's and the lower solid lines are those of the programs transformed by our algorithm.

program	reuse ratio	A runtime	B GC time	B/A	C GC time (reuse)	(B−C)/B	D runtime (flags)	(A−D)/A	E runtime (reuse)	(A−E)/A
`sieve`	85.7%	0.78	0.388	49.7%	0.159	59.0%	0.81	-3.8%	0.89	-14.1%
`merge`	50.0%	0.24	0.172	71.7%	0.106	38.4%	0.26	-8.3%	0.23	4.2%
`qsort`	83.7%	0.67	0.468	69.9%	0.054	88.5%	0.70	-4.5%	0.50	25.4%
`queens`	5.2%	0.33	0.126	38.2%	0.128	-1.6%	0.34	-3.0%	0.35	-6.1%
`msort`	88.6%	0.39	0.212	54.4%	0.048	77.4%	0.40	-2.6%	0.35	10.3%
`mirage`	84.4%	0.21	0.011	5.2%	0.002	81.8%	0.22	-4.8%	0.30	-42.9%
`life`	8.7%	0.49	0.008	1.6%	0.008	0.0%	0.51	-4.1%	0.54	-10.2%
`k-eval`	31.5%	0.41	0.007	1.7%	0.006	14.3%	0.42	-2.4%	0.48	-17.1%
`kb`	3.8%	0.40	0.122	30.5%	0.118	3.3%	0.42	-5.0%	0.43	-7.5%
`nucleic`	17.7%	0.45	0.187	41.6%	0.135	27.8%	0.45	0.0%	0.45	0.0%

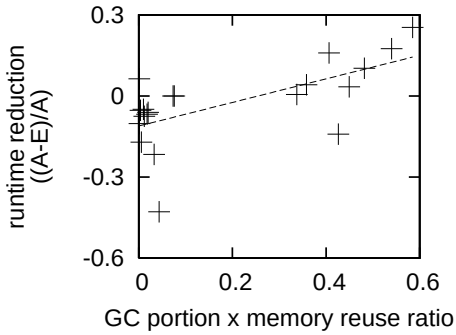
(a) Intel Pentium4 3Ghz, Linux RedHat 9.0

program	reuse ratio	A runtime	B GC time	B/A	C GC time (reuse)	(B−C)/B	D runtime (flags)	(A−D)/A	E runtime (reuse)	(A−E)/A
`sieve`	85.7%	7.15	2.817	39.4%	1.224	56.5%	7.44	-4.1%	7.11	0.6%
`merge`	50.0%	1.69	1.374	81.3%	0.777	43.4%	1.75	-3.6%	1.42	16.0%
`qsort`	83.7%	4.97	3.205	64.5%	0.385	88.0%	5.08	-2.2%	4.10	17.5%
`queens`	5.2%	2.82	0.950	33.7%	0.954	-0.4%	2.99	-6.0%	3.01	-6.7%
`msort`	88.6%	2.96	1.501	50.7%	0.346	76.9%	3.10	-4.7%	2.86	3.4%
`mirage`	84.4%	1.85	0.073	3.9%	0.016	78.1%	2.00	-8.1%	2.25	-21.6%
`life`	8.7%	5.19	0.057	1.1%	0.057	0.0%	5.21	-0.4%	4.86	6.4%
`k-eval`	31.5%	3.78	0.044	1.2%	0.039	11.4%	3.69	2.4%	3.97	-5.3%
`kb`	3.8%	2.85	0.727	25.5%	0.727	0.0%	2.96	-3.9%	2.99	-4.9%
`nucleic`	17.7%	2.70	1.158	42.9%	0.797	31.2%	2.71	-0.4%	2.70	0.0%

(b) Sun UltraSparc 400Mhz, Solaris 2.7



(c) Memory reuse ratio and GC time reduction are co-related.



(d) Runtime reduction is co-related to GC portion × memory reuse ratio.

A seconds: the runtime of Objective Caml 3.06 native code of the original program.
B seconds: the cost of garbage collection of the original program.
C seconds: the cost of garbage collection of the program transformed by our algorithm.
D seconds: the runtime of the program transformed by our algorithm without memory reuses.
E seconds: the runtime of the program transformed by our algorithm.

Figure 4: Runtime changes by memory reuses.

not optimized, we think that it is in general impossible to implement memory reuses cheaper than an allocation in the Objective Caml compiler.

We think that `mirage` becomes slow (142.9% in Pentium4, 121.6% in UltraSparc) because of such expensive implementation of memory reuses. The reuse ratio of `mirage` is high (84.4%) but its GC time is almost nothing (3.9–5.2%). Thus our transformed `mirage` reuses many memory cells resulting in a much accumulated mutable-cell-update overhead that are not offset. For the same reason, `k-eval` and `life` become slow, but their slowdown are not so much as in `mirage` because their reuse ratios are not so high as `mirage`'s.

3.3 Co-relations

- Graph (c) in Figure 2: if our transformation reuses much then it results in much reduction in memory peak. But some exceptions are possible; if reuses focus on those not contributing to the memory peak they can result in small reduction in memory peak.

- Graph (c) in Figure 4: if our transformation reuses much then it results in much reduction in the total garbage collection time.
- Graph (d) in Figure 4: if our transformation reuses much and the portion of the garbage collection time is big, then program's execution time reduction occurs.

4. FUTURE WORK

We are currently implementing the analysis and transformation inside our nML compiler [16] to have it used in daily programming. The main issue in the implementation is to extend our method to handle polymorphism and mutable data structures. To extend our method for polymorphism, we need a sophisticated mechanism for dynamic flags. For instance, a polymorphic function of type $\forall\alpha.\,\alpha \to \alpha$ can take a value with two constructors or one with three constructors. So, this polymorphic input parameter does not fit in the current method because currently we insert twice as many dynamic flags as the number of constructors for each parameter. Our tentative solution is to assign only two flags to the input parameter of type α and to take conjunctions of flags in a call site: when a function is called with an input value with two constructors, instead of passing the four dynamic flags β, β_{ns}, β', and β'_{ns}, we pass $\beta \wedge \beta'$ and $\beta_{\mathrm{ns}} \wedge \beta'_{\mathrm{ns}}$. For mutable data structures, we plan to take a conservative approach similar to that of Gheorghioiu *et al.* [6]: heap cells possibly reachable from modifiable cells cannot be reused.

5. REFERENCES

[1] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proceedings of the European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52, April 2002.

[2] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1995.

[3] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, 1998.

[4] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 262–275, January 1999.

[5] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 70–80, June 2001.

[6] Ovidiu Gheorghioiu, Alexandru Sălcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 273–284, January 2003.

[7] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[8] Williams L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[9] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2001.

[10] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 29–42, 1999.

[11] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Correctness proof on an algorithm to insert safe memory reuse commands. Technical Memorandum ROPAS-2003-19, Research On Program Analysis System, Seoul National University, November 2003. `http://ropas.snu.ac.kr/memo`.

[12] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Inserting safe memory reuse commands into ml-like programs. In *Proceedings of the Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 171–188, San Diego, California, June 2003. Springer-Verlag.

[13] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.06. Institut National de Recherche en Informatique et en Automatique, August 2002. `http://caml.inria.fr`.

[14] Yosuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 271–283, January 1996.

[15] M. Mohnen. Efficient compile-time garbage collection for arbitrary data structures. In *Proceedings of Programming Languages: Implementations, Logics and Programs*, number 982 in Lecture Notes in Computer Science, pages 241–258. Springer–Verlag, 1995.

[16] nML programming language system, version 0.92a. Research On Program Analysis System, Korea Advanced Institute of Science and Technology, March 2002. `http://ropas.snu.ac.kr/n`.

[17] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *The Proceedings of Computer Science and Logic*, pages 1–19, 2001.

[18] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, July 2002.

[19] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–382, March/April 2000.

[20] The Standard ML of New Jersey, version 110.0.7. Bell Laboratories, Lucent Technologies, October 2000. `http://cm.bell-labs.com/cm/cs/what/smlnj`.

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

[22] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). IT University of Copenhagen, April 2002.

`http://www.it-c.dk/research/mlkit`.

[23] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.

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

[25] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *International Conference on Functional Programming and Computer Architecture*, pages 25–28, June 1995.

[26] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland.

[27] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, September 2000.