A New Solution to the Hidden Copy Problem

Deepak Goyal and Robert Paige

New York University, 251 Mercer Street, NY, NY 10012
{deepak, paige}@cs.nyu.edu
Fax # 212-995-4124, Tel # 212 998 3156

Abstract. We consider the well-known problem of avoiding unnecessary costly copying that arises in languages with copy/value semantics and large aggregate structures such as arrays, sets, or files. The origins of many recent studies focusing on avoiding copies of flat arrays in functional languages may be traced back to SETL copy optimization [Schwartz 75]. The problem is hard, and progress is slow, but a successful solution is crucial to achieving a pointer-free style of programming envisioned by [Hoare 75].

We give a new solution to copy optimization that uses dynamic reference counts and lazy copying to implement updates efficiently in an imperative language with arbitrarily nested finite sets and maps (which can easily model arrays, records and other aggregate datatypes). Big step operational semantics and abstract interpretations are used to prove the soundness of the analysis and the correctness of the transformation. An efficient algorithm to implement the analysis is presented. The approach is supported by realistic empirical evidence.

Our solution anticipates the introduction of arbitrarily nested polymorphic sets and maps into JAVA. It may also provide a new efficient strategy for implementing object cloning in Java and object assigment in C++. We illustrate how our methods might improve the recent approach of [Wand and Clinger 98] to avoid copies of flat arrays in a language of first-order recursion equations.

keywords: Copy Optimization, Big Step Operational Semantics, Abstract Interpretation, Must Alias Analysis

1 Introduction

The problem of *hidden copies* arises in languages with *copy/value semantics* and large aggregate structures such as arrays, records, sets, or files. Consider, for example, the following two statements,

```
1        s := t;       -- assign set t to s
2        s with:= x;  -- add element x to s
```

Here, variables `s` and `t` are set-valued and the operation `s with:= x` adds element `x` to set `s`. In a language with reference semantics, the element addition at

Statement 2 above would cause a modification to the value of `t` also. However, such a side effect would be disallowed under copy/value semantics.

Copy/value semantics can be implemented by eager or lazy approaches. In an eager approach, we could implement Statement 1 by assigning a copy of set `t` to `s`; Statement 2 could proceed by adding a copy of the value of `x` into set `s` in place. In a naive lazy approach Statement 1 could assign to `s` a pointer to the body of `t`, which makes `s` and `t` share the same location. Statement 2 could proceed by making a copy of `s`, and augmenting this copy in place with `x`. Making a copy of `s` avoids the side effect of updating other variables that share the location where `s` is stored.

Two interesting strategies have been considered to optimize the lazy approach. One strategy is to use static analysis to prove that an aggregate object is unshared by live variables at a program point, so that the object can be updated destructively at that point. Another is to maintain a dynamic reference count for each location that stores an aggregate object. An object at location L can be updated in place if the reference count at L is 1. We will describe a third strategy with the aim of facilitating more destructive (or in-place) updates than was possible before.

Copy optimization in functional languages with array updates is an important problem of intense current interest [10, 9, 6, 27]. Little seems to be known, however, about Schwartz's extensive investigation of copy optimization for SETL in the 1970's[22]. Since we believe that this early work may be relevant to current research in the area, it is worth summarizing.

1.1 Motivation

SETL[20, 23] is an imperative language with copy/value semantics for assignment and parameter passing, dynamic typing, and built-in finite sets, maps, and tuples of arbitrary depth of nesting. The hidden copy problem has been the major source of inefficiency in two generations of SETL compilers, which use a lazy copy strategy. If such a strategy is not implemented effectively, then hidden copies can potentially degrade program performance from $O(f(n))$ expected time to $O(f(n)^2)$ actual time. Such a slowdown has been actually observed even in small-scale SETL programs[3].

Although the hidden copy problem arises in different languages and language paradigms, it is also crucial to the more general goal, expressed by Hoare [7], of programming without pointers. Without a reasonable solution to this problem, Hoare's ideal cannot be achieved in any practical way. Currently without such a solution, we are forced to choose between two pragmatic, but unsatisfactory, compromises. One of these, exemplified by Ada and C++ is to retain copy/value semantics, but to rely on pointer–oriented programming to obtain efficiency. Another approach, taken by Java, is to assume reference semantics for aggregate objects.

1.2 An unrealized analytic approach

Schwartz developed an interesting but complicated intra-procedural value flow analysis[21, 22] for SETL1[23] in order to detect when destructive updates could be performed. His analysis determined an overestimate of the set of variables that at some level, no matter how deeply embedded, may share the same location. A destructive update to a variable v could be performed if no other variable that might share the location storing the value of v was live. His analysis aimed to be so fine-grained as to detect when destructive updates could be performed on components of aggregate structures.

Sharir [24] showed that value flow analysis did not fit any of the k-bounded monotone dataflow frameworks[26], and conjectured that extensions to interprocedural analysis would be too approximate to be useful. Based on these negative observations, Schwartz's approach was never implemented. Instead, SETL1 implemented a dynamic 'sticky' bit that was initially unset, but was set the first time that a location was shared and subsequently never unset. This solution was completely unsatisfactory, as was demonstrated by the performance of the Ada/Ed compiler[1].

Interestingly, researchers in the functional language community have shown that the kind of may-alias analysis for value flow combined with live variable analysis similar to Schwartz's approach is tractable even in the interprocedural case when datatypes are limited to flat arrays [9, 27]. However, it remains to be seen whether multi-level arrays and other aggregate data structures will confound this approach as they did for SETL.

1.3 A dynamic approach that didn't work

In SETL2[25] dynamic multi-level reference counts for each location storing an aggregate value (tuple, set, or map) are maintained. Highly restricted circularity of pointers ensures that when a location has a reference count greater than 1, then that location is *shared*, and cannot be updated unless it is first copied. Only when a location has reference count 1 can a destructive update be performed.

Since dynamic reference counts only degrade performance by a constant factor, this approach would be worthwhile if it can successfully prevent the loss of asymptotic factors. Unfortunately, the use of dynamic reference counts alone is no solution. The backend of Snyder's SETL2 compiler introduces so many compiler-generated temporary variables (which don't get garbage collected until the end of scope) that practically all data is shared at runtime.

For example, the standard SETL2 code generator implements indexed map assignment

```
f(a)(b) := d
```

using the following lower-level code,

```
t1 := f(a);   -- increment reference count for the location storing f(a)
t1(b) := d;   -- copy f(a) before update
f(a) := t1;   -- in place update if reference count(f) = 1
```

```
s := {}
while P loop
 s with:= x
end loop
c with:= s
```

```
s := {}
while P loop
 { s := copy(s) }
 s with:= x
 c with:= s
end loop
```

```
s := t
{ s := copy(s) }
while P loop
 s with:= x
end loop
c with:= s
d with:= t
```

Fig. 1. Placement of copy operations

which copies the value of map `t1` before it is updated in the second statement. However, if the location L that stores the value of `t` is only shared by `f(a)`, then the reassignment to `f(a)` makes such a hidden copy unnecessary. This problem can be solved dynamically by making `f(a)` undefined by executing the command `f(a) := om` just before the indexed assignment to `t1`. This omega assignment would decrement the reference count at L to 1, and allow a destructive indexed assignment to `t1`.

1.4 Basis for a solution

A solution to the hidden copy problem that assumes dynamic reference counts, may take into account the placement of copy operations and omega assignments (which can decrement dynamic reference counts). Since copy operations can be expensive, it is obvious that their placement needs attention. Figure 1 shows the most desirable place for copy operations to be performed. No hidden copies are needed for the first example. In the second example, a copy operation should be performed just before the element addition to `s` within the while-loop. But in the third example, a single desirable hidden copy should be performed the first time only that the element addition is executed within the while-loop. Our solution is to abide by the SETL2 strategy of making a copy each time an update to an aggregate S cannot be performed destructively (based on a runtime check that the reference count for S is greater than 1). Interestingly, Schwartz's approach (with no dynamic reference counts) would handle the first two examples, but would fail on example 3.

The placement of omega assignments is more subtle. We say that a variable v is *live* at a program point p if the set of live uses of v is non-empty at p [4, 2]. If a variable is not *live* at p, we say it is dead. In order for our transformation to be correct, we will insert omega assignments only to dead variables. Even though the cost of each omega assignment is relatively small, our tests show that the overuse of such assignments leads to suboptimal performance. Thus, our heuristic is to introduce omega assignments only when they are likely to facilitate destructive updates.

In Figure 2, the first example shows that an omega assignment should be placed at a point where a dynamic reference count decrement will actually take place. The second example shows that an omega assignment should be placed at a point where it is not redundant. The third example generalizes the other

```
s := t
-- uses of s
⋮
-- s becomes dead
√
while P loop
 ⊗
 t with:= x
end loop
```

```
while P loop
 s := t
 ⋮
 -- uses of s
 -- s becomes dead
 ⊗
 end loop
√
t with:= x
```

```
while P loop
 if Q then
  ⋮
 else
  s := t
  ⋮
  -- uses of s
  -- s becomes dead
  √
 endif
 ⊗
 t with:= x
 end loop
```

Fig. 2. Placement of `s := om` at $\surd$ is more profitable than at $\otimes$

two examples, and shows that an omega assignment should only be made to a variable that must share the same location as the one being destructively updated.

Suppose that an update to variable s occurs at a program point p, where the value of s is stored at location L. Suppose also that $live(p)$ is the set of variables that are live at p, and that $alias(p, s)$ is the set of variables whose values are all stored at L. Then the update to s cannot be destructive if some variable other than s that belongs to $alias(p, s)$ also belongs to $live(p)$. In such a case as this, it would be futile to perform omega assignments to dead variables at p, since that won't help to reduce the reference count at L to 1. Conversely, if every variable but s that belongs to $alias(p, s)$ is dead, we still can't guarantee that setting all such variables to omega will decrement the reference count at L to 1. This is because an element of a live set-valued variable at p may be stored at location L. But at least we have a good chance. This is what we bet on.

1.5 Outline of the Paper

We report the following new results:

1. An effective copy optimization is given for the first time in a SETL-like language with dynamically typed finite sets and maps of arbitrary depth of nesting. These datatypes can conveniently model a wide range of aggregate datatypes including multilevel arrays and records.
2. In Section 2 we give big step operational semantics [11, 15] for (1) copy/value semantics, (2) a naive lazy copy strategy, and (3) an optimized strategy in which destructive updates are performed on data whose location is unshared. These three semantics are proved to be equivalent.
3. In Section 3 an abstract interpretation[5] is given to support the analysis at each program point of equivalence classes of *variables* that MUST share the

same location. Our analysis is fine-grained enough to detect sharing of map components `f(a)`. A safe approximation of this analysis is computed by an algorithm that runs in time $O(NV^2)$, where N is the number of program points, and V is the number of variables.
4. Finally, in Section 4 the Must-alias analysis is combined with live variable analysis to form an effective intra-procedural transformation that selectively inserts safe omega assignments at appropriate program points according to a heuristic for profitability. A simple but significant test of our approach applied only locally within the SETL2 backend shows ten-fold speedups for realistic large-scale examples.

2 Language

Variables : $s, v, t, f, g, \ldots$
`op` $::= \mathit{with} \mid \mathit{less}$

Expressions:	Commands:
$e ::= \ni v$	$\mathcal{L} ::= \mathit{read}(v)$
$\mid\ v$	$\mid\ v\ :=\ e$
$\mid\ v_1(v_2)$	$\mid\ v_1(v_2)\ :=\ e$ (where v_1 must be distinct from v_2 and e)
$\mid$ `om`	$\mid\ v\ \mathit{op} :=\ e$ (where v must be distinct from e)
$\mid$ `constants`	$\mid\ \mathcal{L}_1; \mathcal{L}_2$
	$\mid$ `if` e `then` $\mathcal{L}_1$ `else` $\mathcal{L}_2$ `endif`
	$\mid$ `while` e `loop` $\mathcal{L}$ `endloop`

Fig. 3. Definition of the language

Figure 3 defines the syntax of the kernel language $\mathcal{L}$ to be transformed by copy optimization. Expression $\ni v$ returns an arbitrary element from set v. Expression $v_1(v_2)$ represents single-valued map application. Commands $v\ \mathit{with} := x$ and $v\ \mathit{less} := x$ are for set element addition and deletion respectively. Without loss of generality we have omitted discussion of a much richer language that can be transformed into $\mathcal{L}$.

2.1 Copy/value Semantics

Figure 4 gives an abstract formulation of the big-step operational copy/value semantics for $\mathcal{L}$. The domain of values is denoted by V_*, and the environment ρ maps variables to $(V_*)_\perp$ (the lifted domain of values), with $\perp$ representing the undefined value. The elements of V_* represent canonical forms which are not described here but which can be obtained easily by lexicographic sorting or multiset discrimination (see [16]). External input constants and program constants are mapped to values in $(V_*)_\perp$ by an external procedure σ. Constants that belong to V_c include integers, sets $\{c_1, \ldots, c_n\}$ of 0 or more constants, smaps $\langle c_1 \mapsto c'_1, \ldots, c_n \mapsto c'_n\rangle$ of 0 or more pairs of constants, and the undefined

Domains

type τ	set V_τ
`int`	$\{int(x) : x = 0, 1, -1, 2, -2, \ldots\}$
`set`	$\{set(s) : s \subseteq V_* \mid \lvert s \rvert < \infty\}$
`smap`	$\{smap(f) : f \subseteq V_* \times V_* \mid \lvert f \rvert < \infty \wedge f$ single-valued$\}$
`*`	$V_{int} \cup V_{set} \cup V_{smap}$

Rules

Environment $\rho : \mathit{Vars} \longrightarrow (V_*)_\perp$
V_c is the set of all syntactically correct constants
Externally defined procedure $\sigma : V_c \longrightarrow (V_*)_\perp$
$domain(u) \stackrel{def}{=} \{x : \exists y \mid [x, y] \in u \quad \wedge \quad y \neq \perp\}$

$$\langle \rho, c \rangle \xrightarrow{e} \sigma(c) \qquad c\text{, a constant} \tag{1}$$

$$\langle \rho, v \rangle \xrightarrow{e} \rho(v) \qquad v\text{, a variable} \tag{2}$$

$$\frac{\langle \rho, v \rangle \xrightarrow{e} set(u), \quad y \in u}{\langle \rho, \ni v \rangle \xrightarrow{e} y} \tag{3}$$

$$\frac{\langle \rho, v \rangle \xrightarrow{e} set(u), \quad u = \{\}}{\langle \rho, \ni v \rangle \xrightarrow{e} \perp} \tag{4}$$

$$\frac{\langle \rho, v_1 \rangle \xrightarrow{e} smap(u_1), \quad \langle \rho, v_2 \rangle \xrightarrow{e} tag(u_2), \quad [tag(u_2), u_3] \in u_1}{\langle \rho, v_1(v_2) \rangle \xrightarrow{e} u_3} \tag{5}$$

$$\frac{\langle \rho, v_1 \rangle \xrightarrow{e} smap(u_1), \quad \langle \rho, v_2 \rangle \xrightarrow{e} tag(u_2), \quad tag(u_2) \notin domain(u_1)}{\langle \rho, v_1(v_2) \rangle \xrightarrow{e} \perp} \tag{6}$$

$$\frac{\langle \rho, c \rangle \xrightarrow{e} u}{\langle \rho, read(v) \rangle \longrightarrow \rho[v \mapsto u]} \qquad \text{where } c \text{ is an external constant} \tag{7}$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} u}{\langle \rho, s := e \rangle \longrightarrow \rho[s \mapsto u]} \tag{8}$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} tag(u), \quad \langle \rho, s \rangle \xrightarrow{e} set(w)}{\langle \rho, s \; op := \; e \rangle \longrightarrow \rho[s \mapsto set(w \; op \; tag(u))]} \tag{9}$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} \perp, \quad \langle \rho, f \rangle \xrightarrow{e} smap(g), \quad \langle \rho, a \rangle \xrightarrow{e} tag(x)}{\langle \rho, f(a) := e \rangle \longrightarrow \rho[f \mapsto smap(Q)]} \qquad \text{where } Q = \{[y, z] \in g \mid y \neq tag(x)\} \tag{10}$$

$$\frac{\langle \rho, e \rangle \xrightarrow{e} tag(u), \quad \langle \rho, f \rangle \xrightarrow{e} smap(g), \quad \langle \rho, a \rangle \xrightarrow{e} tag(x)}{\langle \rho, f(a) := e \rangle \longrightarrow \rho[f \mapsto smap(Q \cup \{[tag(x), tag(u)]\})]} \qquad \text{where } Q = \{[y, z] \in g \mid y \neq tag(x)\} \tag{11}$$

Fig. 4. Copy/value semantics of the Language

atom `om`. Note that $\sigma(\texttt{om}) = \perp$. The initial environment is given by $\rho = \lambda x.\perp$. The semantics of `if-then-else`, `while loop`, and `statement sequence` are straightforward (see [28]), and are omitted.

In rules (1)–(11), judgment $\langle \rho, e\rangle \xrightarrow{e} u$ stands for the evaluation of $\mathcal{L}$ expression e in environment ρ to obtain value $u \in (V_*)_\perp$. Judgment $\langle \rho, c\rangle \longrightarrow \rho'$ represents evaluation of $\mathcal{L}$ command c in environment ρ to obtain new environment ρ'.

In Figure 4 we use the notation $tag(u)$ to represent an element of V_*, $set(u)$ and $smap(u)$ to represent set-valued and smap-valued elements of V_*, and u, u_1, u_2 to represent elements of $(V_*)_\perp$. Under this convention the reader should see that $\perp$ can never be added to a set or smap. Note that the arbitrary selection expression $\ni S$ makes the semantics of $\mathcal{L}$ nondeterministic.

2.2 Lazy Copy Semantics

Whereas the copy/value semantics defines what $\mathcal{L}$ programs mean, we need an abstract formulation of the semantics of lazy copying in order to formally define our copy optimization in Section (3). Figure 5 defines the semantics of $\mathcal{L}$ with a naive lazy copying strategy. As before, we use simple domains to represent canonical forms, with details omitted. Domain V_{loc} is an infinite set of locations where data is stored. In lazy copy semantics environment ρ maps variables either to locations or integers, and *store* γ maps locations either to sets or smaps. In formal terms, $domain(\gamma)$ represents the subset of V_{loc} that has been allocated during execution, and $V_{loc} - domain(\gamma)$ represents unallocated locations. Types `inti`, `seti`, and `smapi` are the implementations of types `int`, `set`, and `smap`, respectively. The implementation of integers remains unchanged, but `seti` is implemented as a set of locations or integers. Similarly `smapi` is implemented as a set of pairs of locations or integers.

We use function *extract* to relate a *location* to the corresponding value that it represents in the copy/value semantics. Function *extract* is recursively defined as,

$$\begin{aligned}
extract(loc(l), \gamma) \;&= case\ \gamma(loc(l))\ of\\
&\qquad seti(s) \Longrightarrow set(\{extract(x) : x \in s\})\\
&\qquad smapi(f) \Longrightarrow smap(\{[extract(x), extract(y)] : [x, y] \in f\})\\
&\qquad end\ case\\
extract(inti(i), \gamma) &= int(i)
\end{aligned}$$

Proposition 1. *Function "extract" is well defined for all locations in store γ at any point in the execution of an $\mathcal{L}$ program*

The proof is a straightforward well-founded induction on the structure of locations in the store, and is omitted. The proof essentially asserts that no sequence of statements can create a set that is a member of itself, or create an smap that is an element of its own domain or range.

Rules (12)–(15) describe how program constants and external constants input by a read statement are evaluated. The method is to allocate an implementation

Implementation Domains

type τ	set V_τ
`loc`	infinite set of atoms loc(l).
`inti`	V_{int}
`seti`	$\{seti(s) : s \subseteq V_{loc} \cup V_{inti} \mid \lvert s\rvert < \infty\}$
`smapi`	$\{smapi(f) : f \subseteq (V_{loc} \cup V_{inti}) \times (V_{loc} \cup V_{inti}) \mid \lvert f\rvert < \infty \wedge f$ single-valued$\}$
`*i`	$V_{inti} \cup V_{seti} \cup V_{smapi}$

Rules

Assume V_c is the set of all the syntactically correct constants.
Environment $\rho : \mathit{Vars} \cup V_c \longrightarrow V_{loc} \cup V_{inti} \cup \{\bot\}$
Store $\gamma : V_{loc} \longrightarrow V_{seti} \cup V_{smapi}$
$\mathit{extract} : V_{loc} \cup V_{inti} \longrightarrow V_*$

$$\langle \gamma, int(x)\rangle \rightsquigarrow (\gamma, inti(x)) \tag{12}$$

$$\frac{l \in V_{loc} - domain(\gamma),\ \langle \gamma_i, u_{i+1}\rangle \rightsquigarrow (\gamma_{i+1}, x_{i+1})\ \forall i = 0, \ldots, n-1}{\langle \gamma_0, set(\{u_1, \ldots, u_n\})\rangle \rightsquigarrow (\gamma_n[l \mapsto seti(\{x_1, \ldots, x_n\})], l)} \tag{13}$$

$$\frac{l \in V_{loc} - domain(\gamma),\ \langle \gamma_i, u_{i+1}\rangle \rightsquigarrow (\gamma_i', x_{i+1}),\ \langle \gamma_i', w_{i+1}\rangle \rightsquigarrow (\gamma_{i+1}, y_{i+1})\ \forall i = 0, \ldots, n-1}{\langle \gamma_0, smap(\{[u_i, w_i] : i = 1, \ldots, n\})\rangle \rightsquigarrow (\gamma_n[l \mapsto smapi(\{[x_i, y_i] : i = 1, \ldots, n\})], l)} \tag{14}$$

$$\frac{\langle \gamma, \sigma(c)\rangle \rightsquigarrow (\gamma', u)}{\langle \rho, \gamma, allocate(c)\rangle \xrightarrow{e} (\rho[c \mapsto u], \gamma')} \tag{15}$$

$$\langle \rho, \gamma, v\rangle \xrightarrow{e} \rho(v) \quad \text{for a variable } v \tag{16}$$

$$\langle \rho, \gamma, c\rangle \xrightarrow{e} \rho(c) \quad \text{for a constant } c \tag{17}$$

$$\frac{\begin{array}{c}\langle \rho, \gamma, v_1\rangle \xrightarrow{e} l_1,\quad \gamma(l_1) = smapi(u_1),\quad \langle \rho, \gamma, v_2\rangle \xrightarrow{e} u_2,\\ ([x, y] \in u_1\ \wedge\ extract(x, \gamma) = extract(u_2, \gamma))\end{array}}{\langle \rho, \gamma, v_1(v_2)\rangle \xrightarrow{e} y} \tag{18}$$

$$\frac{\begin{array}{c}\langle \rho, \gamma, v_1\rangle \xrightarrow{e} l_1,\quad \gamma(l_1) = smapi(u_1),\quad \langle \rho, \gamma, v_2\rangle \xrightarrow{e} u_2,\\ (\forall [x, y] \in u_1 \mid extract(x, \gamma) \neq extract(u_2, \gamma))\end{array}}{\langle \rho, \gamma, v_1(v_2)\rangle \xrightarrow{e} \bot} \tag{19}$$

$$\frac{\langle \rho, \gamma, v\rangle \xrightarrow{e} l,\ \gamma(l) = seti(u),\ y \in u}{\langle \rho, \gamma, \ni v\rangle \xrightarrow{e} y} \tag{20}$$

$$\frac{\langle \rho, \gamma, v\rangle \xrightarrow{e} l,\ \gamma(l) = seti(u),\ u = \{\}}{\langle \rho, \gamma, \ni v\rangle \xrightarrow{e} \bot} \tag{21}$$

Fig. 5. Lazy Copy semantics (cont. on the next page)

$$\frac{\langle \rho, \gamma, allocate(c)\rangle \xrightarrow{e} (\rho', \gamma')}{\langle \rho, \gamma, read(v)\rangle \longrightarrow (\rho[v \mapsto \rho'(c)], \gamma')} \quad \text{where } c \text{ is an external constant} \tag{22}$$

$$\frac{\langle \rho, \gamma, e\rangle \xrightarrow{e} u}{\langle \rho, \gamma, s := e\rangle \longrightarrow (\rho[s \mapsto u], \gamma)} \tag{23}$$

$$\frac{\begin{array}{c}\langle \rho, \gamma, e\rangle \xrightarrow{e} tag(u),\ \langle \rho, \gamma, s\rangle \xrightarrow{e} l,\ \gamma(l) = seti(w), \\ extract(seti(w), \gamma) \neq extract(seti(w\ op\ tag(u)), \gamma),\ l' \in V_{loc} - domain(\gamma)\end{array}}{\langle \rho, \gamma, s\ op := e\rangle \longrightarrow (\rho[s \mapsto l'], \gamma[l' \mapsto seti(w\ op\ tag(u))])} \tag{24}$$

$$\frac{\begin{array}{c}\langle \rho, \gamma, e\rangle \xrightarrow{e} \perp,\ \langle \rho, \gamma, f\rangle \xrightarrow{e} l,\ \langle \rho, \gamma, a\rangle \xrightarrow{e} tag(x), \\ \gamma(l) = smapi(g),\ l' \in V_{loc} - domain(\gamma)\end{array}}{\langle \rho, \gamma, f(a) := e\rangle \longrightarrow (\rho[f \mapsto l'], \gamma[l' \mapsto smapi(Q)])} \tag{25}$$

where $Q = \{[z, y] \in g | extract(z, \gamma) \neq extract(tag(x), \gamma)\}$.

$$\frac{\begin{array}{c}\langle \rho, \gamma, e\rangle \xrightarrow{e} tag(u),\ \langle \rho, \gamma, f\rangle \xrightarrow{e} l,\ \langle \rho, \gamma, a\rangle \xrightarrow{e} tag(x), \\ \gamma(l) = smapi(g),\ l' \in V_{loc} - domain(\gamma)\end{array}}{\langle \rho, \gamma, f(a) := e\rangle \longrightarrow (\rho[f \mapsto l'], \gamma[l' \mapsto smapi(Q \cup [tag(x), tag(u)])])} \tag{26}$$

where $Q = \{[z, y] \in g | extract(z, \gamma) \neq extract(tag(x), \gamma)\}$.

Fig. 5. cont., Lazy Copy Semantics

of a canonical form $\sigma(c)$ of constant c within store γ. We use judgments of the form $\langle \gamma, \sigma(c)\rangle \rightsquigarrow (\gamma', l)$ to say that canonical form $\sigma(c)$ of constant c in the copy/value semantics is allocated into new store γ' at the new location l. Environment ρ maps integer constants to integers, and maps set-valued, or smap-valued constants to corresponding locations. We assume that all program constants are processed by calling a non-$\mathcal{L}$ 'system' command $allocate(c)$ for each constant just before program execution. Thus, the initial environment maps all program constants to corresponding locations or integers.

In Rules (16)–(26), judgment $\langle \rho, \gamma, e\rangle \xrightarrow{e} u$ stands for the evaluation of $\mathcal{L}$ expression e in environment ρ and store γ to obtain either a location or an integer u. Note that preallocation of all program constants before program execution ensures that expression evaluation does not modify either environment ρ or store γ. Judgment $\langle \rho, \gamma, c\rangle \longrightarrow (\rho', \gamma')$ represents the evaluation of $\mathcal{L}$ command c in environment ρ and store γ to obtain new environment ρ' and new store γ'. These rules essentially embody the idea that a simple assignment is implemented by just mapping the left-hand-side variable to the location corresponding to the right-hand-side, but that updates on sets and maps are implemented by modifying a copy. The rule for the read statement $read(v)$ indicates that a new external constant c is input. If c is not an integer, it is allocated within the store at location l say, and the environment is modified so that v is mapped to l.

Although arbitrary selection operator $\ni S$ makes the semantics of $\mathcal{L}$ nondeterministic, we can express the equivalence of copy/value and lazy copy semantics as follows.

$$\frac{\text{Premises of Rule 24} \ \wedge \ \text{not } shared(l,\rho,\gamma)}{\langle \rho,\gamma, s\ op := e\rangle \longrightarrow (\rho, \gamma[l \mapsto seti(w\ op\ tag(u))])} \tag{27}$$

$$\frac{\text{Premises of Rule 25} \ \wedge \ \text{not } shared(l,\rho,\gamma)}{\langle \rho,\gamma, f(a) := e\rangle \longrightarrow (\rho, \gamma[l \mapsto smapi(Q)])} \tag{28}$$

where $Q = \{[z,y] \in g | extract(z,\gamma) \neq extract(tag(x),\gamma)\}$.

$$\frac{\text{Premises of Rule 26} \ \wedge \ \text{not } shared(l,\rho,\gamma)}{\langle \rho,\gamma, f(a) := e\rangle \longrightarrow (\rho, \gamma[l \mapsto smapi(Q \cup [tag(x), tag(u)])])} \tag{29}$$

where $Q = \{[z,y] \in g | extract(z,\gamma) \neq extract(tag(x),\gamma)\}$.

Fig. 6. New rules for optimized lazy copy semantics

Theorem 1. *For given program P, $\langle \lambda x.\bot, P\rangle \longrightarrow \rho_1$ iff $\langle \rho_{init}, \gamma_{init}, P\rangle \longrightarrow (\rho_2,\gamma)$ (where ρ_{init} and γ_{init} are obtained by preallocating the program constants in P) where for all variables v, $\rho_1(v) = extract(\rho_2(v),\gamma)$.*

The proof by rule induction (see for example [28]) is omitted here for the sake of brevity.

2.3 Optimized Lazy Copy Semantics

The semantics of $\mathcal{L}$ with an optimized lazy copying strategy is obtained from the unoptimized semantics by minor modification. This semantics sheds light on how our copy optimization can improve $\mathcal{L}$ programs.

Define a relation $< \subseteq V_{loc} \times V_{loc}$ by the following inductive definition:

$$l_1 < l_2 \overset{def}{\Longleftrightarrow} ((l_1 \in range(\rho) \vee (\exists l_0 | l_0 < l_1)) \wedge \\ ((\gamma(l_1) = seti(s) \wedge l_2 \in s) \vee (\gamma(l_1) = smapi(f) \wedge \exists [x,y] \in f | (x = l_2 \vee y = l_2))))$$

In other words, $l_1 < l_2$ holds if l_1 is reachable from some variable by following a sequence of locations, and either l_1 corresponds to a set having l_2 as a member, or l_1 is a map having l_2 as a member of either its domain or range. For $l \in V_{loc}$, we also define

$var\_share(l,\rho,\gamma) = \{x \in domain(\rho) | \rho(x) = l\}$ `-- variables sharing l`
$set\_share(l,\rho,\gamma) = \{l' : l' < l \ \wedge \ \gamma(l') = seti(\_)\}$ `-- sets sharing l`
$domain\_share(l,\rho,\gamma) = \{l' : l' < l \ \wedge \ \gamma(l') = smap(f) \ \wedge l \in domain(f)\}$
`-- maps whose domain shares l`
$range\_share(l,\rho,\gamma) = \{[d,l'] : l' < l \ \wedge \ \gamma(l') = smap(f) \ \wedge \ f(d) = l'\}$
`-- maps whose range shares l at specific domain points`

Then the extent to which location l is shared in environment ρ and store γ is defined by reference count,

$$refcount(l,\rho,\gamma) \overset{def}{=} (|var\_share(l,\rho,\gamma)| + |set\_share(l,\rho,\gamma)| + \\ |domain\_share(l,\rho,\gamma)| + |range\_share(l,\rho,\gamma)|).$$

Predicate $shared(l, \rho, \gamma) \overset{def}{\Longleftrightarrow} refcount(l, \rho, \gamma) > 1$ decides whether l is shared, and is used in the optimized lazy copy semantics to allow data stored at unshared locations to be updated destructively.

The new semantics includes Rules (12)-(26) (from Figure 5), adds new Rules (27)-(29) (see Figure 6), and conjoins premise $shared(l, \rho, \gamma)$ to Rules(24)-(26).

Proposition 2. *Optimized lazy copy semantics preserves the semantics of unoptimized lazy copy semantics.*

In the next section we describe an alias analysis that computes a close approximation to sets

$$var\_share(l, \rho, \gamma) \cup \{y : \exists l \in range(\rho) | y \in range\_share(l, \rho, \gamma)\},$$

in order to justify copy optimization.

3 Must-Alias Analysis

This section describes the static data flow analysis used to compute a safe (sound) approximation to the must-alias relation $R \subseteq Vars \times Vars$ at each program point. A pair $\langle x, y \rangle$ belongs to R at a program point p, if we can guarantee that $\rho(x) = \rho(y)$ (wrt lazy copy semantics) for all possible environments ρ reaching program point p in any execution of the program. Note that the absence of a pair $\langle x, y \rangle$ from R does **not** imply that $\rho(x) \neq \rho(y)$ for all environments ρ reaching p. Note that $\langle x, y \rangle \notin R$ does not even imply the existence of at least one environment ρ reaching p such that $\rho(x) \neq \rho(y)$, since the relation R is only a safe approximation.

It is easy to see that any *must-alias* relation R is an equivalence relation. So R can be efficiently implemented as a *partition* of the set of variables $Vars$. P is a partition of $Vars$ *iff* P is a set of nonempty mutually disjoint subsets of $Vars$ whose union is all of $Vars$. The next subsection states some of the properties of partitions that will be useful in our analysis. A much more detailed description of these and many other properties can be found in [13].

3.1 Notation

Elements of a partition are sometimes called *blocks*. Let $\mathcal{P}(S)$ denote the set of all partitions over a finite set S. If $Q, P \in \mathcal{P}(S)$, then we say that Q is a *refinement* of P, denoted by $Q \sqsubseteq P$, *iff* $\forall b \in Q | (\exists b' \in P | b \subseteq b')$. The partially ordered set $(\mathcal{P}(S), \sqsubseteq)$ has maximum element $\{S\}$ and minimum element $\{\{v\} : v \in S\}$, and, being finite, has the finite-descending-chain condition. The set of partitions form a Lattice, for which the meet ($\sqcap$) is defined by

$$P \sqcap Q \overset{def}{=} \{b_P \cap b_Q : b_P \in P, b_Q \in Q | b_P \cap b_Q \neq \{\}\ \}.$$

Figure 7 gives an example of the meet operation.

$$\boxed{v_1, v_2, v_3}\ \boxed{v_4, v_5, v_6} \quad \sqcap \quad \boxed{v_1, v_2}\ \boxed{v_3, v_4, v_5}\ \boxed{v_6}$$

$$= \quad \boxed{v_1, v_2}\ \boxed{v_3}\ \boxed{v_4, v_5}\ \boxed{v_6}$$

Fig. 7. Example of the $\sqcap$ (meet) operation for the Partition Lattice

Each partition on a finite set S can be represented as a single-valued map from S to names of blocks, where each block in the partition is associated with a unique name. Let $name : blocks \rightarrow names$ map distinct blocks to distinct names. Then, the implementation $P' : S \rightarrow names$ of a partition P is given by,

$$P' = \cup_{b \in P} \{[x, name(b)] : x \in b\} \tag{30}$$

In the following sections, we will abuse our notation by associating the symbol b with both the block b (*i.e.* a subset of S) and $name(b)$. Thus, equation(30) could be rewritten as

$$P' = \cup_{b \in P} \{[x, b] : x \in b\}.$$

Given an implementation P of a partition of set S, $P(x)$ denotes the name of the block containing element x, and $[x]_P$ denotes the equivalence class of element x (*i.e.* the set of elements in the same block as x). Of course, since changing the name of any block does not change the underlying partition, we define a notion of equivalence between implementations of partitions by saying that two such implementations P and Q are equivalent if their underlying partitions are equal.

The following notation for overriding is very useful in describing partitions that are constructed from existing partitions by moving elements from one block, either to another existing block, or to a new block. Partition $P/\{[x, P(y)]\}$ denotes the partition obtained by moving element x to the block containing element y. Similarly, the partition obtained by moving x to a new block by itself is denoted by $P/\{[x, new(block)]\}$, where $new(block)$ returns a new block name. This notation for overriding can be extended to describe the movement of more than one variable into other existing or new blocks. Let $X : A \rightarrow names$ be single-valued, with $A \subseteq S$. Then, P/X represents a partition obtained from P by moving elements in A into either existing or new blocks, and we have $P/X \equiv P'$, where $P'(y) = P(y)$ if $y \notin A$ and $P'(y) = X(y)$ otherwise. Finally, for $X_1 : A \rightarrow names$, and $X_2 : B \rightarrow names$ single-valued, with $A \cap B = \phi$, $X_1 \uplus X_2$ denotes the obvious union of these two maps. The disjointness of A and B ensures the single-valuedness of the resulting union.

3.2 Abstract Interpretation

This section describes an abstract interpretation approach based closely along the lines of [18], for computing a sound must-alias relation at each program point. As described in the previous subsection, the alias relation is a partition of the set of variables in the program. The set *Vars* comprises of all variables of the form v and $f(v)$ (smap application) that appear textually in the program. We use a nonstandard definition of *Vars* by including the variables of the form $f(v)$ appearing explicitly in the program. This enables us to maintain information such as whether $\langle f(v), s\rangle$ belongs to the must-alias relation or not. In fact, this is a key factor that enables us to do a reasonable must-alias analysis for arbitrarily nested sets and maps. Note, that even though only single-level map applications of the form $f(v)$ can appear textually in an $\mathcal{L}$ program, multi-level map applications such as $f(x)(y)(z)$ or $f(g(h(x)))$ are easily translated into $\mathcal{L}$.

We consider an abstract interpretation framework where the environment-store pairs (from the lazy copy semantics) form the concrete domain and partitions form the abstract domain. We also define an abstraction function, that takes an element $\langle\rho,\gamma\rangle$ of the concrete domain to an element P of the abstract domain. The abstraction function merely states that two variables are mapped into the same block if and only if they are mapped to the same location in $\langle\rho,\gamma\rangle$.

$$\text{Abstraction Function } \beta : \mathit{Env} \;\times\; \mathit{Store} \rightarrow \mathit{Partition}$$

$$\begin{aligned}&\beta(\langle\rho,\gamma\rangle) = P \iff \forall v_1, v_2 \in \mathit{Vars} :\\ &(\langle\rho,\gamma,v_1\rangle \xrightarrow{e} l_1 \wedge \langle\rho,\gamma,v_2\rangle \xrightarrow{e} l_2 \wedge (l_1 = l_2) \iff P(v_1) = P(v_2))\end{aligned} \quad (31)$$

Just as statements act as (Env, Store) transformers in the concrete domain, similarly, statements act as partition transformers in the abstract domain. Figure 8 defines the abstract semantics for the statements in $\mathcal{L}$. We can read $\langle P, st\rangle \xrightarrow{a} P'$ as saying that the statement st transforms partition P to P'. The abstract semantics are most easily understood by looking at the examples in Figure 9.

Suppose variables f and g are pointing to the same location. Then, for any variable u, $\rho(f(u))$ and $\rho(g(u))$ (if defined) will evaluate to the same location. In other words, if maps f and g are equal, then so are $f(u)$ and $g(u)$. Therefore, we make sure that the partitions in our analysis satisfy a congruence property that says that *if f and g are in the same block, then f(u) and g(u) should be in the same block.*

We now proceed with a step-by-step explanation of the rules for the abstract semantics described in Figure 8.

1. For statements of the form $read(s)$, $s\ op := e$, $s :=\ni v$:
   The only variables whose values could be affected by these statements are of the form s, or $s(v)$, or $g(s)$. The effect of overriding P by $\{[s, new(block)]\}$ is to move s to a new block (see example Figure 9). Also, we see in the example that, if f and g were equal before the read, the fact that $read(s)$ cannot modify either f or g, guarantees that f and g (and, hence $f(s)$ and $g(s)$) must be equal after the read. This congruence effect is captured by

$$\uplus_{b\in P}(\text{let } n = new(block)\text{in } \{[g(s), n] : g \in m_1\{s\} \cap b\}),$$

$$\langle P, st \rangle \xrightarrow{a} P'$$

$$\begin{aligned}
\text{Notation: } & m_1\{s\} = \{f \in \mathit{Vars} | f(s) \in \mathit{Vars}\} \\
& m_1[S] = \cup_{s \in S} m_1\{s\} \\
& m_2\{f\} = \{s \in \mathit{Vars} | f(s) \in \mathit{Vars}\} \\
& m_2[S] = \cup_{f \in S} m_2\{f\} \\
& m_f[S] = \{v \in \mathit{Vars} | f(v) \in S\}
\end{aligned}$$

Case 1. If st is of the form $read(s)$, $s\ op := e$, or $s :=\ni v$:

$$\begin{aligned}
P' = P/\ & \{[s, new(block)]\} \uplus \\
& \uplus_{b \in P} (\text{let } n = new(block) \text{ in}\{[s(v), n] : v \in m_2\{s\} \cap b\}) \uplus \\
& \uplus_{b \in P} (\text{let } n = new(block) \text{ in}\{[g(s), n] : g \in m_1\{s\} \cap b\})
\end{aligned}$$

Case 2. If st is of the form $s := t$:

$$\begin{aligned}
P' = P/\ & \{[s, P(t)]\} \uplus \\
& \uplus_{b \in P | b \cap m_1[[t]_P] \neq \phi} \text{ let } v =\ni \{v \in [t]_P | m_1\{v\} \cap b \neq \phi\} \text{ in} \\
& \qquad \text{let } f =\ni m_1\{v\} \cap b \text{ in}\{[g(s), P(f(v))] : g \in m_1\{s\} \cap b\} \uplus \\
& \uplus_{b \in P | b \cap m_2[[t]_P] \neq \phi} \text{ let } v =\ni \{v \in [t]_P | m_2\{v\} \cap b \neq \phi\} \text{ in} \\
& \qquad \text{let } f =\ni m_2\{v\} \cap b \text{ in}\{[s(g), P(v(f))] : g \in m_2\{s\} \cap b\} \uplus \\
& \uplus_{b \in P | b \cap m_1[[t]_P] = \phi} (\text{let } n = new(block) \text{ in}\{[g(s), n] : g \in m_1\{s\} \cap b\}) \uplus \\
& \uplus_{b \in P | b \cap m_2[[t]_P] = \phi} (\text{let } n = new(block) \text{ in}\{[s(g), n] : g \in m_2\{s\} \cap b\})
\end{aligned}$$

Case 3. The case for $s := f(t)$ is obtained from Case 2 by substituting t by $f(t)$.
Case 4. If st is of the form $f(t) := s$, then

$$\begin{aligned}
P' = P/\ & \{[f, new(block)]\} \uplus \\
& \{[f(v), P(s)] : v \in m_2\{f\} \cap [t]_P\} \uplus \\
& \uplus_{b \in P | b \neq P(t) \wedge b \cap m_f[[s]_P] = \phi} (\text{let } n = new(block) \text{ in}\{[f(v), n] : v \in m_2\{f\} \cap b\}) \uplus \\
& \uplus_{b \in P} (\text{let } n = new(block) \text{ in}\{[g(f), n] : g \in m_1\{f\} \cap b\})
\end{aligned}$$

Fig. 8. Abstract Semantics

which says that for all variables in $\{g \in b | g(s) \in \mathit{Vars}\}$, we should move the corresponding variables of the form $g(s)$ into a block by themselves. A similar rule applied to variables of the form $s(v)$ and $s(u)$.

2. Statements of the form $s := t$:

   The rule for this statement is more interesting. The effect of $\{[s, P(t)]\}$ is to move s into the block containing t. Suppose there exist variables v and $f(v)$ in the program such that $v = t$ before the assignment. Then after the assignment, we know that $s = t$. Therefore, if the variable $f(s)$ exists, it must equal $f(v)$ (by the congruence condition). The complicated rule shown in Figure 8 essentially captures this effect by considering two cases. In the first case, there exist variables v and $f(v)$ such that ($v \in [t]_P$), so we move $f(s)$ to the block containing $f(v)$. In the second case, no such v exists, so $f(s)$ is moved to a new block. Again, we have to be careful in the second

$$\boxed{s,\, t}\ \boxed{f,\, g}\ \boxed{f',\, g'}\ \boxed{f'(t),\, g'(t),\, y}\ \boxed{f(s),\, g(s),\, f(t),\, g(t),\, x}$$

read(s)

$$\boxed{s}\ \boxed{t}\ \boxed{f{,}g}\ \boxed{f',\, g'}\ \boxed{f'(t),\, g'(t){,}y}\ \boxed{f(s),\, g(s)}\ \boxed{f(t),\, g(t){,}x}$$

t := s

$$\boxed{s,\, t}\ \boxed{f,\, g}\ \boxed{f',\, g'}\ \boxed{f'(t),\, g'(t)}\ \boxed{y}\ \boxed{f(s),\, g(s),\, f(t),\, g(t)}\ \boxed{x}$$

$$\boxed{t,\, u}\ \boxed{v_1}\ \boxed{v_2}\ \boxed{s,\, f(v_1),\, g(v_1)}\ \boxed{f,\, g}\ \boxed{f(v_2),\, g(v_2)}\ \boxed{f(t),\, g(t),\, f(u),\, g(u),\, x}$$

f(t) := s

$$\boxed{t,\, u}\ \boxed{v_1}\ \boxed{v_2}\ \boxed{s,\, f(v_1),\, g(v_1),\, f(t),\, f(u)}\ \boxed{f}\ \boxed{g}\ \boxed{f(v_2)}\ \boxed{g(v_2)}\boxed{g(t),\, g(u),\, x}$$

Fig. 9. Illustrative examples of the abstract semantics

case to maintain congruence. The variables of the form $s(u)$ are handled in a similar manner.

3. Statements of the form $f(t) := s$:
   Clearly this statement modifies the value of map f, which is moved to a new block. Looking at our example in Figure 9, we see that since t and u are equal, we move both $f(t)$ and $f(u)$ to the block containing s. If v_2 is not in the same block as t, this does not mean that $s \neq t$. A modification to $f(t)$ could conceivably cause a modification to $f(v_2)$. This is why $f(v_2)$ moves into a block by itself. However, $f(v_1)$ is not moved to a block by itself, because it was already equal to s; the statement $f(t) := s$ could not have possible changed its value to something other than s. The rule given in Figure 8 takes care of all these subtleties.

We now state a number of theorems dealing with the soundness of the abstract semantics. The proofs are long and can not be included here because of space constraints.

Theorem 2 (Local Soundness). *If st is a simple assignment statement, then*

$$\langle \rho, \gamma, st\rangle \longrightarrow \langle \rho', \gamma'\rangle \mathit{and}\ \langle \beta(\langle \rho, \gamma\rangle), st\rangle \xrightarrow{a} P' \ \Rightarrow\ (P' \sqsubseteq \beta(\langle \rho', \gamma'\rangle))$$

Simply put, if the incoming partition is a safe approximation of the incoming environment-store pair, then the outgoing partition is also a safe approximation of the outgoing environment-store pair.

Theorem 3 (Monotonicity). *For any simple assignment statement st, and partitions P and Q such that $P \sqsubseteq Q$, we have*

$$\langle P, st\rangle \xrightarrow{a} P' \mathit{and} \ \langle Q, st\rangle \xrightarrow{a} Q' \Rightarrow \ P' \sqsubseteq Q'$$

We define a path p from *start* (the beginning of the program) to a program point i as any sequence of simple assignment statements that lead from *start* to i. Note that, if there exists a cyclic path to i, then there are infinitely many possible paths to i (corresponding to 0 or more loops around the cycle). Let

$$\mathit{paths}(i) = \{\text{All paths to program point } i\}.$$

Furthermore, let us define f_p and f_p^* to be the concrete and abstract transfer functions for path p, *i.e.,*

$$f_p(\langle \rho, \gamma\rangle) = \langle \rho', \gamma'\rangle \ \overset{def}{\iff} \ \langle \rho, \gamma, p\rangle \longrightarrow \langle \rho', \gamma'\rangle, \mathit{and}$$

$$f_p^*(Q) = Q' \ \overset{def}{\iff} \ \langle Q, p\rangle \xrightarrow{a} Q'$$

Then the *Meet-over-all-paths solution, i.e.,* $\sqcap_{p \in paths(i)} \beta(f_p(\langle \rho_{init}, \gamma_{init}\rangle)$ is a safe approximation to the must-alias relation.

Theorem 4 (Global Soundness). *For all program points i in the program,*

$$\sqcap_{p \in paths(i)} f_p^*(\beta(\langle \rho_{init}, \gamma_{init}\rangle)) \ \sqsubseteq \ \sqcap_{p \in paths(i)} \beta(f_p(\langle \rho_{init}, \gamma_{init}\rangle)$$

In other words, the solution to the data flow analysis problem in the abstract domain is a safe approximation of the solution in the concrete domain.

The Proof follows from Local Soundness (Theorem 2) and Monotonicity (Theorem 3).

3.3 Algorithm

Let the program points be numbered from $0, \ldots, n$, where program point 0 corresponds to the start of the program. We use $st(j, i)$ to denote the statement between program point j and its successor i. We use the results from [12] to claim that the meet-over-all-paths solution is given by the greatest fixed point solution of the following equation:

$$\begin{aligned} F(P_0, \ldots, P_n) = (&g_0(P_0, \ldots, P_n), \\ &g_1(P_0, \ldots, P_n), \\ &\vdots \\ &g_n(P_0, \ldots, P_n)) \end{aligned} \tag{32}$$

$$\text{where } g_0(P_0, \ldots, P_n) = \beta(\langle \rho_{init}, \gamma_{init}\rangle)$$
$$\text{and } g_i(P_0, \ldots, P_n) = \sqcap_{j \in pred(i)} f^*_{st(j,i)}(P_j), \ \ \forall i = 1, \ldots, n.$$

The above greatest fixed point can be computed by Kildall's iterative algorithm [14] (shown in Figure 10).

Let V denote $|\mathit{Vars}|$ and let N denote the number of nodes in the control flow graph.

$$\begin{array}{ll}\text{Init}: & \forall i = 0, \ldots n \;\; F(i) = \top \\ \text{Loop}: & \textit{repeat} \\ & \forall i = 1, \ldots, n \;\; F(i) = \sqcap_{j \in pred(i)} f^*_{st(j,i)}(F(j)) \\ & \text{until all } F(i)\text{'s stabilize.}\end{array}$$

Fig. 10. Kildall's Iterative algorithm for greatest fixed point computation

Proposition 3. *1. Given partitions P_1 and P_2, we can compute $P_1 \sqcap P_2$ in $O(V)$ time.*
2. Given a partition P, we can compute $f^*_{st}(P)$ for any simple assignment in $O(V)$ time.

(1) can be proven directly from [13]. (2) has a direct implementation using simple data structuring and the techniques in [13].

Proposition 4. *Each iteration of Kildall's algorithm can be implemented in $O(N \times V)$ time.*

This follows directly from Proposition 3 assuming that the control flow graph has been processed so that each node has at most 2 predecessors and at most 2 successors [17].

Proposition 5. *Kildall's algorithm converges in at most $N \times V$ iterations. Thus the time complexity of the iterative algorithm is $O(N^2 \times V^2)$.*

The proof relies on the fact that the length of a strictly decreasing chain of partitions is bounded by V and that each iteration decreases the value of $F(i)$ at least at one program point i.

The time complexity of Kildall's algorithm can be improved by using a worklist strategy. The idea is to maintain a worklist of program points i that do not satisfy the condition $F(i) = \sqcap_{j \in pred(i)} f^*_{st(j,i)}(F(j))$. Figure 11 describes the modified algorithm.

Proposition 6. *The complexity of the worklist algorithm in Figure 11 is $O(N \times V^2)$.*

The proof depends on showing that each program point can be added to the worklist at most V times. Furthermore, each time a program point is removed from the worklist, the processing takes $O(V)$ time. Hence the time complexity is $O(N \times V^2)$.

4 Copy Optimization Transformation

The problem of live variable analysis is well understood (see [19], for example). We have a live variable analysis that takes map variables such as $f(v)$

$worklist = \{1, \ldots, n\}$
$\forall i = 1, \ldots, n \;\; F(i) = \top$
while ($worklist \neq \phi$) loop
    remove an arbitrary element x from the $worklist$
    $F(x) = \sqcap_{y \in pred(x)} f^*_{st(y,x)}(F(y))$
    for $z \in succ(x)$ loop
        if $(z \notin worklist) \wedge (\sqcap_{w \in pred(z)} f^*_{st(w,z)}(F(w)) \sqsubseteq F(z))$ then
            $worklist\ with := z$
        endif
    end loop
end loop

Fig. 11. Algorithm based on Worklist strategy

into account. This analysis is also proven correct using the framework of abstract interpretation. The abstract semantics and the proof of correctness are straightforward and are omitted.

The final solution is obtained as follows. For each update assignment, we look at the must-alias set for the variable being updated, and if all the other variables in this set are dead, these are assigned omegas just before the update. We are investigating improvements to this strategy in order to facilitate elimination of more copies.

5 Applications

Destructive array update optimization is critical for writing scientific codes in functional languages. Recently, Wand and Clinger [27] propose a solution for a call-by-value functional language based on interprocedural flow analysis for aliasing and liveness. Their optimization is based on the idea that if a *sound* live variable analysis can ensure that the array on which the update is being performed is not live after the update, then the update can be performed destructively.

Consider the definitions of a few functions in the simple first-order functional language considered by Wand and Clinger in Figure 12. Their analysis tries to prove that the location corresponding to array B is always dead at the time when B is updated. However, they will not be successful in doing so for the simple reason that this is not true. This can be seen from the fact that the array A is live after the first call to function f from inside function g. Consequently, all updates on B will result in the creation of a new copy, although, if the calls to f are from function h, then the destructive updates on the arrays would still be legal. This problem is very reminiscent of the problems that made Schwartz's value flow analysis [22] ineffective for solving the copy optimization problem in Setl.

We believe that the key idea of using dynamic reference counts together with alias and liveness analysis could contribute to an effective solution to the

fun $Sum(A) = \ldots$ //sum the elements of array A
fun $f(B, i, j) = Sum(update(B, i, j))$
fun $g(A, i_1, i_2) = f(A, i_1, 0) + f(A, i_2, 0)$
fun $h(A, i_1) = f(A, i_1, 0)$

Fig. 12. Small functional program fragment illustrating difficulties of a static analysis based approach to copy optimization

problem. If the program fragment in Figure 12 is implemented in an imperative language with copy/value semantics, then the use of reference counts, an interprocedural live variable analysis, and placement of dynamic reference count decrements would ensure that the reference count of array B is 1 when function f is called by function h, thereby allowing a destructive update.

6 Conclusions and Future Work

This paper presents a new approach to copy optimization that trades potentially asymptotic costs of hidden copies for a constant factor overhead to maintain dynamic reference counts. The analysis and transformation are proved correct using formal semantics and abstract interpretations. A new low polynomial time algorithm is given to carry out the analysis.

It would be interesting to extend this work to the interprocedural case. We also want to explore partition refinement strategies found in [8] and [13] to improve the algorithm. It would be interesting to see how further optimization could cut down or eliminate dynamic reference counts. Our analysis deals with arbitrarily nested sets and maps that can be used to simulate a wide variety of datatypes such as records or even pointers (by the use of single-valued maps called *ref* and *deref*). We believe that our analysis and algorithmic techniques may apply to pointer analysis.

References

1. ADA UK News, Vol. 6, No. 1, pp. 14–15, Jan 1985.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.
3. J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993.
4. J. Cocke and J. Schwartz. *Programming Languages and Their Compilers.* Lecture Notes. Courant Institute, New York University, New York, 1969.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proc. Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
6. M. Draghicescu and S. Purushotham. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118:231–262, 1993.

7. C. A. R. Hoare. Data reliability. In *Proc. of the Intl. Conf. on Reliable Software*, pages 528–533, 1975.
8. J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. Proc. Intl. Symp. on Theory of Machines and Computation.
9. P. Hudak. A semantic model of reference counting and its abstraction. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming*, pages 351–363. ACM, 1986.
10. P. Hudak and A. Bloss. Avoiding copying in functional and logic programming languages. In *Conference record of the 12th Annual ACM Symposium on Principles of programming languages*, pages 300–314. ACM, 1985.
11. G. Kahn. Natural semantics. In *Proc. STACS'87*. Springer-Verlag, 1987. Lecture Notes in Computer Science, Vol. 247.
12. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
13. J. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10), 1995.
14. G. A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.
15. H. R. Nielson and F. Nielson. *Semantics with Applications, A formal introduction*. Wiley, 1992.
16. R. Paige and Z. Yang. High level reading and data structure compilation. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 456–469, Paris, France, 15–17 Jan. 1997.
17. B. K. Rosen, M. Wegman, and K. Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.
18. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
19. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.
20. J. Schwartz. *On Programming: An Interim Report on the SETL Project, Installments I and II*. New York University, New York, 1974.
21. J. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec. 1975.
22. J. Schwartz. Optimization of very high level languages, parts I, II. *J. of Computer Languages*, 1(2-3):161–218, 1975.
23. J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
24. M. Sharir. A few cautionary notes on the convergence of iterative data-flow analysis algorithms. Setl Newsletter 208, New York University, April 1978.
25. K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University, 1990.
26. R. E. Tarjan. A unified approach to path problems. *JACM*, 28(3):577–593, July 1981.
27. M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages*. IEEE, May 1998.

28. G. Winskell. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1994.