

Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates

A.V.S. Sastry, William Clinger and Zena Ariola
Department of Computer Science
University of Oregon
Eugene, OR, 97403
email: [sastry, will, ariola]@cs.uoregon.edu

Abstract

The aggregate update problem in functional languages is concerned with detecting cases where a functional array update operation can be implemented destructively in constant time. Previous work on this problem has assumed a fixed order of evaluation of expressions. In this paper, we devise a simple analysis, for strict functional languages with flat aggregates, that derives a good order of evaluation for making the updates destructive. Our work improves Hudak’s work [14] on abstract reference counting, which assumes fixed order of evaluation and uses the domain of sticky reference counts. Our abstract reference counting uses a 2-point domain. We show that for programs with no aliasing, our analysis is provably more precise than Hudak’s approach (even if the fixed order of evaluation chosen by Hudak happens to be the right order). We also show that our analysis algorithm runs in polynomial time. To the best of our knowledge, no previous work shows polynomial time complexity. We suggest a technique for avoiding excessive copying even in those cases where the analysis determines that an update cannot be made destructively. We have implemented the algorithm and tested it on some common example programs. The results show that a good choice of the order of evaluation determined by the analyzer indeed makes most of the updates destructive.

1 Introduction

The array data structure poses an implementation problem in functional languages. Because of the semantics of these languages, an update operation generally requires creation of a new copy of the entire array; the update at the appropriate index is made in the new copy. The old copy needs to be kept intact because there may be other subcomputations in the program that refer to it. This straightforward implementation of the update operator leads to inefficient use of memory, and degrades the time complexity of an algorithm in proportion to the size of the largest array.

In this paper we present an update analysis algorithm for first-order strict functional languages with flat aggregates. The analysis, in contrast to previous work [14], does not assume any fixed order of evaluation of arguments of any

function. The problem with fixing an order a priori is that several opportunities for destructive updating may be lost. Moreover, we show that our algorithm has polynomial time complexity. None of the previous works on strict languages [8, 14, 12] give polynomial time complexity bounds.

We present a simple abstract reference counting analysis that exploits the syntactic information available in a program for determining the liveness of a variable. Our abstract reference count domain uses a 2-point domain. We show that for programs without aliasing our analysis is provably more precise than Hudak’s abstract reference counting [14].

In the cases where the analysis determines that an update cannot be made destructive, we present a simple heuristic for copy avoidance.

We have designed and implemented an abstract interpreter to perform the update analysis. Our results show that for most examples, a good order of evaluation makes all the updates destructive, whereas an analysis that assumes a fixed order detects only the updates that can be optimized with that order.

The rest of the paper is organized as follows. Section 2 describes the language chosen for analysis. Section 3 presents an overview of the solution with some notation used in later sections. Section 4 describes the abstract domains and the abstract semantic functions used in the analysis. Section 5 shows how to derive a good evaluation order using the information obtained from these abstract functions. The abstract reference count analysis, which uses the order of evaluation derived previously, is described in Section 6. In Section 7 we show that our analysis algorithm runs in polynomial time. In Section 8 we present our experimental results. In Section 9 we compare our work with Hudak’s abstract reference counting. Section 10 describes a simple heuristic of copy avoidance by judicious copy introduction. We conclude the paper with a discussion of related work and with future directions.

2 The Source and Intermediate Languages

We consider a first-order, call-by-value language (Figure 1) with flat aggregates, that is, an aggregate can only contain non-aggregate values. In our language nested function definitions are not permitted. This is not a serious restriction because it is always possible to eliminate all nested function definitions by “lambda lifting” [15]. To simplify the analysis we will work with an intermediate language, where each subexpression is given a unique name which can be thought

c	\in	$Cons$	Constants
x	\in	V	Variables
op	\in	$Prims$	Primitive functions (<i>i.e.</i> $+$, $-$, \dots)
f	\in	F	User Defined Functions
e	\in	Exp	$::=$ $c \mid x \mid op(e_1, \dots, e_n)$ $\mid Sel(e_1, e_2)$ $\mid Upd(e_1, e_2, e_3)$ $\mid f(e_1, \dots, e_n)$ $\mid \text{If } e_0 \text{ then } e_1 \text{ else } e_2$
pr	\in	$Program$	$::=$ $\{f_1 x_{1_1} \dots x_{k_1} = e_1;$ $\quad \vdots$ $\quad f_n x_{1_n} \dots x_{k_n} = e_n\}$

Figure 1: The Syntax of the Source Language

se	\in	SE	$::=$ $c \mid x \mid t_i$
e	\in	$IExp$	$::=$ $se \mid op(se_1, \dots, se_n)$ $\mid Sel(se_1, se_2)$ $\mid Upd(se_1, se_2, se_3)$ $\mid f(se_1, \dots, se_n)$ $\mid \text{If } se \text{ then } e_1 \text{ else } e_2$ $\mid \text{Let } [t_1 = e_1, \dots, t_n = e_n] \text{ In } t_i \text{ End}$
pr	\in	$IProg$	$::=$ $\{f_1 x_{1_1} \dots x_{m_1} = e_1;$ $\quad \vdots$ $\quad f_n x_{1_n} \dots x_{m_n} = e_n\}$

Figure 2: The Syntax of the Intermediate Language

of as a compiler generated temporary variable [3]. As we will see shortly, the analysis will distinguish these temporary variables from other identifiers. The syntax of the intermediate language is given in Figure 2. Notice that the only way an expression appears inside another expression is through a conditional or a let-expression. The scope of a let-binding $t_i = e_i$ in a let-expression $\text{Let}[\dots, t_i = e_i, \dots, t_n = e_n] \text{ In } t \text{ End}$ consists of all the occurrences of t_i in expressions e_{i+1} to e_n and t .

The select operator Sel takes an aggregate and an index and returns the value stored at that index in the aggregate. The update operator Upd takes an aggregate a , an index i , and a value v and returns a new aggregate which contains v at the index i but is otherwise like a .

A program is a set of mutually recursive definitions, where each function's body is a closed expression.

3 Overview

Assuming a fixed order of evaluation limits the extent of the analysis, as shown in the following example:

$$\begin{aligned}
 f \ x \ i &= \text{If } i = 0 \text{ then } x \text{ else } Upd(x, i, i) \\
 g \ x \ i \ j &= Sel(x, i) + Sel(f(x, i), i * j) + Sel(x, 2i)
 \end{aligned}$$

where any fixed order of evaluation of the $+$ operator cannot make the update in f destructive. However, by choosing to evaluate the first occurrence of $+$ from left to right and the second occurrence from right to left the update can be done destructively. The aim of our analysis is to first find such an order. An update can then be converted into a destructive update if it updates an aggregate which is no longer live, that is, there does not exist any further references to it.

A suitable ordering is obtained by forcing the evaluation of those expressions which select an aggregate before the evaluation of those expressions that update that aggregate. To that end, we associate with each expression the aggregates that are selected and updated (*Selects-and-Updates analysis*). For example, given the expression $Sel(x, i)$ we will associate the information that the aggregate x is selected by that expression. For $Sel(t_1, i)$, where $t_1 = f(x, y)$, the information depends on which aggregates are returned by the function f . In particular, we are interested in knowing if the function f returns any of its arguments. For example, if f returns a new aggregate and f contains an update of x then that update can be done destructively. If f returns the variable x then the update will not be done destructively. The *propagation analysis* will collect the information regarding which variables are returned or propagated by the evaluation of an expression. For example, the variables propagated by the expression $g(x, y)$, where g simply returns x , consist of x , if x and y are not aliased. Therefore, the aliasing information has to be computed in order for the propagation information to be accurate. In conclusion, in order to devise a good order of evaluation we need to collect the following information: propagation of variables, aliasing and variables selected and updated.

To collect all of the above information we will use the technique of abstract interpretation [7, 1]. We can then augment the dependency graph associated with an expression with additional edges, which are called *interference edges*. An expression e_2 is said to *interfere* with another expression e_1 if e_2 updates an aggregate that is selected or updated by e_1 . Any destructive update in e_2 before the evaluation of e_1 is not permissible because it would change the semantics of the expression e_1 . The augmented graph is used to derive an order of evaluation for the expressions.

In the following we assume that all variables in a program are distinct. Moreover, given a binding $t_i = e_i$, we assume the existence of a function $\text{expr-of}(t_i)$ which, given the temporary variable t_i , returns the corresponding expression, *i.e.* e_i . We assume that the reader is familiar with partial orders, the least upper bound (lub) operation which is denoted by \sqcup , fixpoints etc. We avoid the use of subscripts for the domain of the least upper bound operation when it is clear from the context. We use fix as the least fixpoint operator. All the abstract domains are finite and the abstraction functions are monotonic, therefore least fixpoints exist and are computable [19].

Environments are finite maps from the syntactic domain of identifiers to some other domain of interest. The empty environment, which is the least element in the domain of environments, is denoted by \perp . The value of an identifier x in an environment σ is represented as $\sigma[x]$. The environment obtained by extending another environment σ with a binding $x \mapsto v$ is written as $\sigma[x \mapsto v]$. An environment mapping the variables x_1 to v_1, \dots, x_n to v_n is written as $[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$. The notation $[f_i \mapsto e_i]$ stands for $[f_1 \mapsto e_1, \dots, f_n \mapsto e_n]$. The least upper bound operation on the domain of environments can be defined in terms of

the least upper bound operation on the range of the environments. If $Env = Ide \rightarrow D$, the lub of two environments env_1 and env_2 is defined as

$$env_1 \sqcup_{Env} env_2 = \lambda x \in Ide. env_1[x] \sqcup_D env_2[x]$$

4 Abstract Functions

We define three abstract domains and abstraction functions. Each abstraction gives one particular meaning to the program capturing one particular property of interest.

The function \mathcal{H} computes the variables propagated by an expression; \mathcal{A} computes the aliasing of formal parameters in a program; \mathcal{S} computes the sets of aggregates selected and updated in an expression evaluation. These three abstractions are used for defining three functions \mathcal{H}_p , \mathcal{A}_p , and \mathcal{S}_p , that compute three abstract meanings $\mathcal{H}_p[[pr]]$, $\mathcal{A}_p[[pr]]$, $\mathcal{S}_p[[pr]]$ for a program pr .

4.1 Propagation Analysis

The abstract domains needed for propagation analysis are shown in Figure 3.a, where V represents the set of all distinct variables in a program, that is, V does not include the temporary variables. The abstraction function \mathcal{H} (see Figure 3.b) takes an expression, a variable and a function environment and returns the set of variables propagated by that expression.

Notice that while the set of variables propagated by an identifier is obtained by looking up the variable environment, the set of variables propagated by a temporary variable is obtained by computing the set of variables propagated by its associated expression. Since the primitive operators of our language do not propagate any of their arguments, the set of variables propagated by the primitive expression is empty. As our language does not permit non-flat aggregates, a select expression does not propagate any variable. The set of variables propagated by an update expression is also empty because semantically the update operation returns a new aggregate which is different from any of the aggregates bound to any of the variables appearing in its arguments. For a function call, the sets of variables propagated by each actual parameter is computed recursively. The abstraction of the function, looked up from the function environment, is applied to the abstract values of the actual parameters.

The function \mathcal{H} is used for defining \mathcal{H}_p which takes a program pr and returns an environment in which each user defined function is mapped to an abstract function that gives the information about the variables propagated by the body of the function. Notice that we do not build a table to represent the input-output behavior of a function, as shown in the following example. Consider the function below:

$$f \ x \ y \ z = \text{If } g(x) \text{ then } x \ \text{else } f(y, z, x)$$

its \mathcal{H} -meaning is defined as :

$$f \ x \ y \ z = x \cup f(y, z, x)$$

The least fixpoint of the functional associated with the above equation is computed by successive approximations [19]. The sequence of these successive approximations will be :

$$\begin{aligned} f^0 &= \lambda x y z. \emptyset \\ f^1 &= \lambda x y z. x \cup \emptyset \\ f^2 &= \lambda x y z. x \cup y \\ f^3 &= \lambda x y z. x \cup y \cup z \end{aligned}$$

V		Program Variables
F		User Defined Functions
D	$= \mathcal{P}(V)$	Powerset Domain over V
$\sigma \in VEnv$	$= V \rightarrow D$	Variable Environments
$\rho \in FEnv$	$= F \rightarrow D^n \rightarrow D$	Function Environments

Figure 3.a: Domains for Propagation Analysis

$$\mathcal{H} : IExp \rightarrow VEnv \rightarrow FEnv \rightarrow D$$

$$\begin{aligned} \mathcal{H}[[c]]\sigma \rho &= \emptyset \\ \mathcal{H}[[x]]\sigma \rho &= \sigma[x] \\ \mathcal{H}[[t_i]]\sigma \rho &= \mathcal{H}[[\text{expr-of}(t_i)]]\sigma \rho \\ \mathcal{H}[[op(se_1, \dots, se_n)]]\sigma \rho &= \emptyset \\ \mathcal{H}[[\text{Sel}(se_1, se_2)]]\sigma \rho &= \emptyset \\ \mathcal{H}[[\text{Upd}(se_1, se_2, se_3)]]\sigma \rho &= \emptyset \\ \mathcal{H}[[f_k(se_1, \dots, se_n)]]\sigma \rho &= \rho[f_k](\mathcal{H}[[se_1]]\sigma \rho, \dots, \mathcal{H}[[se_n]]\sigma \rho) \\ \mathcal{H}[[\text{If } se_0 \text{ then } e_1 \text{ else } e_2]]\sigma \rho &= \mathcal{H}[[e_1]]\sigma \rho \cup \mathcal{H}[[e_2]]\sigma \rho \\ \mathcal{H}[[\text{Let } [t_1 = e_1, \dots, t_n = e_n] \text{ In } t_i \text{ End}]]\sigma \rho &= \mathcal{H}[[t_i]]\sigma \rho \end{aligned}$$

Figure 3.b: The function \mathcal{H}

$$\mathcal{H}_p : IProg \rightarrow FEnv$$

$$\mathcal{H}_p[[pr]] = \text{fix}(\lambda \rho. \rho[f_i \mapsto \lambda y_1, \dots, y_{m_i}. \mathcal{H}[[e_i]]_{[x_1 \mapsto y_1, \dots, x_{m_i} \mapsto y_{m_i}]} \rho])$$

Figure 3.c: The function \mathcal{H}_p

where f^3 , the fixpoint, conveys the information that all the parameters of f can be propagated.

The formal definition of the function \mathcal{H}_p is given in Figure 3.c. We refer to $\mathcal{H}_p[[pr]]$ of a program pr as the propagation environment.

4.2 Aliasing Analysis

The aliasing information can be represented as an environment in which each variable is bound to a set of variables consisting of its aliases. The domains necessary for computing aliasing information of a program are given in Figure 4.a. The lub operation on the domain of aliasing environments is defined in terms the lub (the set union operation) operation of the domain D . The abstraction function \mathcal{A} (see Figure 4.b) takes an expression, an aliasing environment, and a function environment and returns a new aliasing environment.

The only expression that can cause aliasing is the function call. For a function call, we determine the variables propagated by its actual parameters using the aliasing environment as the variable environment, and the propagation environment as the function environment. For each pair of formal parameters of the function, we determine if the sets of variables propagated by the corresponding actual parameters are disjoint. If the two sets are disjoint, then no aliasing is caused by the particular function call. If the sets are not disjoint, the two formal parameters of the function can

$D = \mathcal{P}(V)$ Power set of Variables
 $\sigma \in AEnv = V \rightarrow D$ Aliasing Environments

Figure 4.a: Domains for Aliasing Analysis

$\mathcal{A} : IExp \rightarrow AEnv \rightarrow FEnv \rightarrow AEnv$

$\mathcal{A}[\text{c}] \sigma \rho = \sigma$
 $\mathcal{A}[\text{x}] \sigma \rho = \sigma$
 $\mathcal{A}[\text{t}_i] \sigma \rho = \sigma$
 $\mathcal{A}[\text{op}(se_1, \dots, se_n)] \sigma \rho = \sigma$
 $\mathcal{A}[\text{Sel}(se_1, se_2)] \sigma \rho = \sigma$
 $\mathcal{A}[\text{Upd}(se_1, se_2, se_3)] \sigma \rho = \sigma$
 $\mathcal{A}[\text{f}_k(se_1, \dots, se_n)] \sigma \rho =$
 let
 $\quad v_i = \mathcal{H}[se_i] \sigma \rho$
 $\quad a_{ij} = v_i \cap v_j, 1 \leq i, j \leq n, i \neq j$
 in
 $\quad \sigma \sqcup [x_{i_k} \mapsto (\sigma[x_{i_k}] \cup \sigma[x_{j_k}]) \mid a_{ij} \neq \emptyset,$
 $\quad \quad x_{i_k} \in (\sigma[x_{i_k}] \cup \sigma[x_{j_k}])]$
 end
 $\mathcal{A}[\text{if } se_0 \text{ then } e_1 \text{ else } e_2] \sigma \rho = \mathcal{A}[e_1] \sigma \rho \sqcup \mathcal{A}[e_2] \sigma \rho$
 $\mathcal{A}[\text{Let } [t_1 = e_1, \dots, t_n = e_n] \text{ ln } t_i \text{ End}] \sigma \rho = \bigsqcup_{i=1}^n \mathcal{A}[e_i] \sigma \rho$

Figure 4.b: The function \mathcal{A}

potentially be aliased. We thus update the aliasing information of the parameters of the called function. For example, if the formal parameters x_{i_k} and x_{j_k} could be aliased then the aliases of x_{i_k} and x_{j_k} have to be updated with a new alias-set. The new alias-set is the union of the aliases of x_{i_k} and x_{j_k} . Notice that we treat aliasing as a transitive relation. If two different calls of a function alias the first and second arguments and the first and third arguments, respectively, we assume that all the three formal parameters of that function are aliased to one another. This imprecision can be avoided at the expense of increased complexity of the aliasing analysis.

The \mathcal{A} -meaning of a program is computed by the function \mathcal{A}_p as defined in Figure 4.c. In this definition, σ_{id} is the identity environment in which every variable is bound to a singleton set containing itself. For a program pr , $\mathcal{A}_p[[pr]]$ is the aliasing information of all the user-defined functions in the program.

4.3 Selects-and-Updates Analysis

The domains needed for selects-and-updates analysis are given in Figure 5.a. The first component of the abstract domain D_{su} represents the set of variables that are possibly selected in the evaluation of the expression. The second component gives the set of variables possibly updated by the expression evaluation. The domain $SEnv$ represents the abstraction of each user defined function to a function that returns the set of variables selected and updated by the function, given the set of aggregates bound to each of its ar-

$\mathcal{A}_p : IProg \rightarrow AEnv$

$\mathcal{A}_p[[pr]] =$
 $\text{let } \rho = \mathcal{H}_p[[pr]]$
 in
 $\quad fix(\lambda \sigma. (\bigsqcup_{i=1}^n \mathcal{A}[e_i] \sigma \rho \sqcup \sigma_{id}))$
 end

Figure 4.c: The function \mathcal{A}_p

guments. We use the abstract propagation environment and the function \mathcal{H} for determining the set of aggregates selected and updated by an expression.

The semantic functions \mathcal{S} and \mathcal{S}_p , are given in Figure 5.b and Figure 5.c, respectively.

5 Deriving an Order of Evaluation

Our objective is to choose an order of evaluation of the bindings of a let-expression that allows us to perform the updates destructively. Given a let expression $\text{Let } [t_1 = e_1, \dots, t_n = e_n] \text{ ln } t_i \text{ End}$, we first construct a dependency graph, whose nodes are the expressions t_1, \dots, t_n . We say an expression t_j depends on t_i , if t_i appears in the free variables of t_j . In the dependency graph, this dependence is represented as a directed edge (i,j), indicating that the node i must be evaluated before the node j. The dependency graph will necessarily be a directed acyclic graph (dag) because we are assuming strict semantics and our language doesn't allow cyclic data structures.

We then augment the dependency graph with additional edges which represent the imposed precedence of expression evaluation. We call these additional edges *interference-edges*. An interference edge (i,j) conveys the information that e_j possibly updates an aggregate needed by e_i . The graph so obtained is called precedence graph. The *interferes* predicate is defined in Figure 6. We find the strongly connected components of the precedence graph using the algorithm given in [2]. We construct a new graph whose nodes are the strongly connected components of the precedence graph. There is an edge E_{ij} between the nodes V_i and V_j if $\exists v_k \in V_i$ and $v_l \in V_j$ such that (k,l) is an edge of the precedence graph. The new graph is necessarily a dag. A topological sorting of the new dag gives an order for the evaluation of the strongly connected components of the precedence graph. An ordering of all the expressions is then obtained by replacing each component by any ordering of its elements.

The complexity of deriving an order for a let-expression containing n bindings is $O(n^2)$. This bound cannot be improved because all pairs of let-bindings have to be considered for interference.

In the final ordered let-expression, $\text{Let } [t_1 = e_1, \dots, t_k = e_k] \text{ ln } t_i \text{ End}$, the expression t_i is evaluated before t_k . Given the order of expression evaluation, one can then determine the set of live variables at each binding, as explained in the next section.

	$D_{su} = \mathcal{P}(V) \times \mathcal{P}(V)$	Selected and Updated Variables	$interferes\ e_i\ e_j = let$
$\sigma \in VEnv = V \rightarrow \mathcal{P}(V)$		Variable Env.	$fenv = \mathcal{H}_p[[pr]]$
$\rho \in FEnv = F \rightarrow \mathcal{P}(V)^n \rightarrow \mathcal{P}(V)$		Propagation Env.	$aenv = \mathcal{A}_p[[pr]]$
$\delta \in SEnv = F \rightarrow \mathcal{P}(V)^n \rightarrow D_{su}$		Selects-and-updates Env.	$suenv = \mathcal{S}_p[[pr]]$
			$\langle s_i, u_i \rangle = \mathcal{S}[[e_i]]suenv\ fenv\ aenv$
			$\langle s_j, u_j \rangle = \mathcal{S}[[e_j]]suenv\ fenv\ aenv$
			<i>in</i>
			$u_j \cap (s_i \cup u_i) \neq \emptyset$
			<i>end</i>

Figure 5.a: Domains for Selects-and-Updates Analysis

$\mathcal{S} : IExp \rightarrow SEnv \rightarrow FEnv \rightarrow VEnv \rightarrow D_{su}$

$$\begin{aligned}
\mathcal{S}[[c]]\delta\ \rho\ \sigma &= \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[[x]]\delta\ \rho\ \sigma &= \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[[op(se_1, \dots, se_n)]]\delta\ \rho\ \sigma &= \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[[Sel(se_1, se_2)]]\delta\ \rho\ \sigma &= \langle \mathcal{H}[[se_1]]\sigma\ \rho, \emptyset \rangle \\
\mathcal{S}[[Upd(se_1, se_2, se_3)]]\delta\ \rho\ \sigma &= \langle \emptyset, \mathcal{H}[[se_1]]\sigma\ \rho \rangle \\
\mathcal{S}[[If\ se_0\ then\ e_1\ else\ e_2]]\delta\ \rho\ \sigma &= \mathcal{S}[[e_1]]\delta\ \rho\ \sigma \sqcup \mathcal{S}[[e_2]]\delta\ \rho\ \sigma \\
\mathcal{S}[[f_k(se_1, \dots, se_n)]]\delta\ \rho\ \sigma &= \delta[f_k](\mathcal{H}[[se_1]]\sigma\ \rho, \dots, \mathcal{H}[[se_n]]\sigma\ \rho) \\
\mathcal{S}[[Let\ [t_1 = e_1, \dots, t_n = e_n]\ in\ t_i\ End]]\delta\ \rho\ \sigma &= \bigsqcup_{i=1}^n \mathcal{S}[[e_i]]\delta\ \rho\ \sigma
\end{aligned}$$

Figure 5.b: The function \mathcal{S}

$\mathcal{S}_p : IProg \rightarrow SEnv$

$$\mathcal{S}_p[[pr]] = fix(\lambda\delta. \delta[f_i \mapsto \lambda y_1, \dots, y_{m_i}. \mathcal{S}[[e_i]]\ \delta\ (\mathcal{H}_p[[pr])], [x_1 \mapsto y_1, \dots, x_{m_i} \mapsto y_{m_i}])$$

Figure 5.c: The function \mathcal{S}_p

6 Abstract Reference Count Analysis

Our abstraction of reference counts is a 2-point domain R whose least element 1 represents the existence of exactly one reference to an aggregate and \top represents the existence of multiple references. Intuitively, a variable is *live* at a program point if there are any future references to it. Depending on where the reference occurs we will distinguish between *local* and *global* liveness. For our analysis a binding is a program point. Consider the following example:

```

f x i = Let
  [t1 = g(x, i);
   t2 = Sel(t1, i);
   t3 = Sel(x, i);
   t4 = +(t2, t3);]
  In t4
  End;
g y j = Let
  [t = Upd(y, i, i)]
  In t
  End;

```

In the body of f , we will say that x is locally live at point t_1 because there exists another reference to x , namely at point

Figure 6: The predicate *interferes*

t_3 . In the body of g , y is not locally live because there are no further references to y in g after t . However, y is globally live because there exists a call to g (i.e. $g(x, i)$) with a live actual parameter. The global liveness is computed by the semantic function \mathcal{R} , which returns the abstraction of reference counts of objects bound to the formal parameters of a function in all possible calls that could arise in any program execution.

The abstract domains and the abstract semantic function \mathcal{R} for computing the abstract reference environment are shown in Figures 7.a and 7.b, respectively. In the definition of \mathcal{R} we make use of the functions $FV(e_j)$ and $Vars(e_j)$. $FV(e_j)$ returns the set of free variables in the expression e_j , and $Vars(e_j)$ is $FV(e_j) \setminus \{t_1, \dots, t_n\}$.

The definition of \mathcal{R} needs explanation only for the function call and the let-expression. Given a function call, we determine the set of variables propagated by each actual parameter of the function using \mathcal{H} . The liveness of each parameter is tested by checking if it is globally or locally live. Global liveness of an actual parameter is tested by checking if at least one of the variables (or its aliases) propagated by the actual parameter has the value \top in the reference environment $renv$. Local liveness is tested by determining if at least one of the variables (or its aliases) propagated by the actual parameter is in the live variable set (*lset*), which is given as an argument to \mathcal{R} .

In the case of a let-expression, the set of live variables at each binding is computed and the bindings are analyzed recursively. The live variables at a binding are the variables, and their aliases, that appear in the expressions yet to be evaluated; the variables propagated by already evaluated expressions which are used in some expression which is yet to be evaluated; and the set of variables live (*lset*) after the evaluation of the whole let-expression.

The \mathcal{R} -meaning of a program, which is an abstract reference environment, is computed by the function \mathcal{R}_p . For a program pr , the definition of \mathcal{R}_p is given in Figure 7.c.

Given $\mathcal{R}_p[[pr]]$, it is easy to decide if an update can be performed destructively. Consider an update expression, $t_i = \text{Upd}(x, y, z)$. Suppose that $lset_i$ is the set of variables that are live at point i . This update cannot be made destructively if there is at least one variable propagated by x which is either globally or locally live. This condition is formally expressed as follows:

$$\exists y \in (\mathcal{H}[[x]]\ \mathcal{A}_p[[pr]]\ \mathcal{H}_p[[pr]]) \text{ such that } \mathcal{R}_p[[pr]][y] = \top \text{ or } y \in lset_i$$

$FEnv$	$= F \rightarrow \mathcal{P}(V)^n \rightarrow \mathcal{P}(V)$	
$AEnv$	$= V \rightarrow \mathcal{P}(V)$	
R	$= \{1, \top\}$	Reference Counts
$REnv$	$= V \rightarrow R$	Reference Env.
$LSet$	$= \mathcal{P}(V)$	Live Variables

Figure 7.a: Domains for Reference Count Analysis

$\mathcal{R} : IExp \rightarrow REnv \rightarrow FEnv \rightarrow AEnv \rightarrow LSet \rightarrow REnv$

$\mathcal{R}[[c]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[x]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[t_i]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[op(se_1, \dots, se_n)]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[Sel(se_1, se_2)]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[Upd(se_1, se_2, se_3)]] \text{ renv fenv aenv lset} = \text{renv}$
 $\mathcal{R}[[\text{If } se_0 \text{ then } e_1 \text{ else } e_2]] \text{ renv fenv aenv lset} =$
 $\quad \mathcal{R}[[e_1]] \text{ renv fenv aenv lset} \sqcup \mathcal{R}[[e_2]] \text{ renv fenv aenv lset}$
 $\mathcal{R}[[f_k(se_1, \dots, se_n)]] \text{ renv fenv aenv lset} =$
 $\quad \text{let } v_1 = \mathcal{H}[[se_1]] \text{ aenv fenv}$
 $\quad \vdots$
 $\quad v_n = \mathcal{H}[[se_n]] \text{ aenv fenv}$
 in
 $\quad \text{renv} \sqcup [x_{j_k} \mapsto \top \mid \exists x \in v_j, \text{renv}[x] = \top \vee x \in \text{lset}]$
 end
 $\mathcal{R}[[\text{Let } [t_1 = e_1, \dots, t_n = e_n] \text{ In } t_i \text{ End}]] \text{ renv fenv aenv lset} =$
 let
 $\quad \text{lset}_i = (\cup \{ \text{aenv}[x] \mid x \in \bigcup_{j=i+1}^n \text{Vars}(e_j) \}) \cup$
 $\quad (\cup \{ \mathcal{H}[[t_k]] \text{ aenv fenv} \mid k < i, \exists j, i < j \leq n, t_k \in \text{FV}(t_j) \})$
 $\quad \cup \text{lset}$
 in
 $\quad \prod_{i=1}^n \mathcal{R}[[e_i]] \text{ renv fenv aenv lset}_i$
 end

Figure 7.b: The function \mathcal{R}

Notice that we can use the same analysis for introducing explicit deallocation instructions, that is, we can safely deallocate an array if it is no longer live.

7 Complexity Analysis

In this section, we derive the complexity of the functions \mathcal{H}_p , \mathcal{A}_p , \mathcal{S}_p and \mathcal{R}_p . Each of these functions involves a fixpoint calculation. The complexity bound is estimated by giving a bound on the maximum number of iterations needed for the fixpoint computation and the complexity of each iteration. The program size is represented in terms of three parameters: n , the number of functions in the program; k , the maximum function arity; and m , the number of non-let-expressions in the program, where $m \geq n$. The number of functions and the maximum function arity are used for obtaining a bound on the number of iterations needed in a fixpoint computation. The number of non-let-expressions in the program along with the function arity is used for obtaining the work done in each iteration. The basic unit of analysis for each of the abstract functions is a non let-expression.

$\mathcal{R}_p : IProg \rightarrow REnv$
 $\mathcal{R}_p[[pr]] =$
 $\quad \text{let } \text{fenv} = \mathcal{H}_p[[pr]]$
 $\quad \quad \text{aenv} = \mathcal{A}_p[[pr]]$
 $\quad \text{in}$
 $\quad \quad \text{fix}(\lambda \sigma. \prod_{i=1}^n \mathcal{R}[[e_i]] \sigma \text{ fenv aenv } \emptyset)$
 $\quad \text{end}$

Figure 7.c: The function \mathcal{R}_p

A let-expression is analyzed by analyzing its bindings. The work in any iteration can be bounded by finding the complexity of analyzing a non-let expression and multiplying it by the number of non-let expressions in the program.

\mathcal{H}_p computes an element of $FEnv$ in which each function symbol is bound to an abstract propagation function, which can only be a union of some its formal parameters. The number of iterations needed for computing the fixpoint is kn . The work done in each iteration is bounded by $O(mk^2)$, therefore the complexity of \mathcal{H}_p is $O(mnk^3)$.

Each variable can be aliased to at most k variables as we consider only aliasing among the formal parameters of a function. Therefore the number of iterations needed for computing the aliasing environment is $O(nk^2)$. We also show that the work done in each iteration is also bounded by $O(mk^3)$ which gives us the overall complexity of $O(mnk^5)$ for the function \mathcal{A}_p .

\mathcal{S}_p computes an element of $SEnv$ where each function symbol is mapped to an abstract selects-and-updates function. Any selects-and-updates function is a pair whose components are unions of some of arguments of the functions. If the maximum arity of a function is k , then the maximum size of a chain of pairs of sets is bounded by k^2 . Thus the number of iterations needed for the fixpoint computation is $O(nk^2)$. The complexity of one iteration can be shown to be $O(mk^2)$ which gives us the complexity of \mathcal{S}_p as $O(mnk^4)$.

The number of variables in the reference environment is $O(nk)$. Each variable takes values from the 2-point domain R . The number of iterations in the fixpoint point computation is bounded by nk because each variable starts with a value 1 and at least one variable becomes \top in each iteration. It can be shown that the complexity of computing liveness of an actual parameter is $O(k)$; it gives a bound of $O(mk^2)$ on the time for each iteration. The complexity of \mathcal{R}_p is $O(mk^3)$. The details of the complexity analysis are available in [17].

8 Results

We have implemented the above algorithm in Standard ML and tested several examples that use flat aggregates. Two dimensional arrays are represented as one dimensional arrays. The programs chosen are gaussian elimination, matrix transpose, matrix multiplication, LU-decomposition, quick-sort, bubble sort, counting sort¹, array initialization and two artificial programs c_1 and c_2 . The results of the analysis are shown in Figure 8.

¹the range of numbers is known

Program	No. of upds	No. of destructive upds		
		no ordering	ltr	rtl
gauss-elm-1	5	5	4	5
gauss-elm-2	5	5	5	4
transpose	2	2	1	2
matmul	3	3	2	3
LU-decomp	2	2	2	2
recursive-fft	4	4	4	4
qsort	4	4	4	4
bubblesort	2	2	1	2
count-sort	4	4	3	4
init	1	1	1	1
c_1	2	2	0	1
c_2	1	0	0	0

Figure 8: Performance of the Update Analysis Algorithm

We show the number of updates that are converted into destructive updates under various ordering strategies. The first column of the table shows the total number of update operators in the program. Our results are shown in the column with heading ‘no ordering’. The next two columns show the results of the analysis with the left-to-right and right-to-left ordering, respectively.

The three programs `matmul`, `transpose` and `bubblesort`, use the function `swap` which is defined as follows:

$$\text{swap } a \ i \ j = \text{Upd}(\text{Upd}(a, i, \text{Sel}(a, j)), j, \text{Sel}(a, i))$$

To make these updates destructive, the arguments of the update operator have to be evaluated from right to left. Our analysis derives this order whereas Bloss [4] assumes that the `Upd` operator is evaluated from right to left. Similarly, in the gaussian elimination program, which takes arrays `A` and `B` and computes `X` such that `AX = B`, the order of evaluation of expressions in recursive calls is important for destructive updating. Deriving the order of evaluation relieves the user of thinking about the order in which one should pass the arguments to a function to make the program efficient. We wrote the same function with different orderings of the formal parameters and the analyzer finds an appropriate ordering in each case. These two different versions correspond to the two entries `gauss-elm-1` and `gauss-elm-2` in the figure. For the programs `quicksort`, `init`, and `recursive-fft`, any evaluation order is good for destructive updating. Our analysis is also able to find an ordering that interleaves the evaluation of arguments of two different functions, as shown in the following program c_1 :

$$\begin{aligned} f \ x \ y \ i &= \text{Sel}(x, y + i); \\ g \ x \ y \ i &= f(\text{Upd}(y, i, i), \text{Sel}(x, i), \text{Sel}(y, 2 * i)) + \\ &\quad f(\text{Upd}(x, i, i), \text{Sel}(y, i), \text{Sel}(x, 2 * i)); \end{aligned}$$

In this example, both the updates can be performed destructively only if all the selects are evaluated before the updates. For the following program c_2 :

$$\begin{aligned} f \ x \ i &= \text{If } i = 0 \text{ then } x \text{ else } \text{Upd}(x, i, 2i) \\ h \ y \ i \ j &= \text{Sel}(f(y, i), j) + \text{Sel}(f(y, j), i) \end{aligned}$$

there is no ordering which makes the update destructive. Our analysis safely concludes that the update cannot be made destructive.

9 Comparison with Hudak’s Work

Hudak described an abstraction of reference counting for update analysis in [14]. In this section, we show that our analysis is more precise. We first summarize Hudak’s approach and show the sources of imprecision and discuss how we avoid these imprecisions.

Hudak’s work defines an abstract store semantics of a first-order language in which the reference count operation is modeled as a side-effect. This is precisely the reason why he fixes an order of evaluation a priori. The store represents the abstraction of reference count of each object. When a function is called, the reference count of its actual parameter is incremented by the total number of occurrences of the corresponding formal parameter in the body of the function. In the body of the function when a variable is encountered, the reference count is decremented, mimicking exactly the actual execution. With a finite domain of sticky reference counts with a maximum reference count $maxrc$, the increment and decrement operations become imprecise when the reference count reaches the value $maxrc$.

We show that our analysis is more precise by first considering a program with an update which can be converted into a destructive update which is detected by our analysis whereas Hudak’s approach fails to detect it. Suppose the sticky reference count domain has the maximum reference count r (a fixed number). Consider the following program

$$\begin{aligned} f \ x \ i \ j &= g(x) + \text{Sel}(\text{Upd}(x, i, i), j) \\ g \ y &= e \end{aligned}$$

Suppose the body of g has r occurrences of y with no updates of y . When $g(x)$ is called from f , x has a reference count 2. By the initialization of Hudak’s approach, the reference count of x becomes $(2 - 1 + r) = \infty$ just before the execution of the body of g . If a variable gets a reference count of ∞ it stays there. Therefore, the reference count at the update is ∞ indicating that the update cannot be made destructive.

Our analysis would correctly determine that the update can be made destructive because by our abstraction, $g(x)$ does not propagate x which means that all the occurrences of x created by the call would be consumed after the execution of the body of g . The update has the last reference to x , therefore it can be made destructive.

Now we show that for programs with no aliasing, our approach is as precise as Hudak’s approach. Suppose our approach marks an update as non-destructive. There are two possibilities. Suppose one of the variables propagated by the first argument of the update operator is locally live. It means that there is at least one occurrence of that variable in the rest of the body of the function. It follows that the reference count of the object bound to that variable has to be at least 2 just before the update, therefore Hudak’s method would conclude that the update cannot be made destructive.

The second possibility is that one of the propagated variables by the first argument of the update operator is globally live. It means there is one instance, in which the object bound to the variable was locally live in some function f which eventually calls the function containing the update operator. It implies that the object bound to the variable has a reference count of least 2 (1 for the occurrence of the live occurrence of the variable in the body of f and the other for the reference held by the update operator). In this case also, Hudak’s analysis would conclude that the update cannot be made destructive.

One source of imprecision in Hudak’s approach is because of the association of objects to expressions generating them. Two different objects generated by different instances of an update are assumed to be same. This imprecision can be reduced by better approximation of the domain of abstract locations. But the better approximations would increase the height of the domain of abstract locations. In our approach we represent the objects by program variables thus avoiding the problem.

10 Copy avoidance by judicious copying

Can we reduce some copying in those cases where the analysis determines that an update cannot be performed destructively? We use a simple heuristic that if possible, an update appearing in the body of a recursively defined function should be made destructive by explicit copying of the arguments to the function call arising from some other function. Consider the following example:

```

rev a n      =  rev1(a, a, 0, n);
rev a b i n  =  if i = n then a
                else
                rev1(Upd(a, i, sel(b, n ⊔ ⊥i)), b, i+1, n);

```

In this example, any update analysis algorithm has to safely conclude that the update has to be non-destructive because of the aliasing of the arguments a and b . This algorithm has a cost of $O(n^2)$ in both time and space. We know that the update could have been made destructive if a was not aliased to b by the call from rev . In other words, we have to determine whether the update in $rev1$ can be made destructive if it is called appropriately (i.e. with no aliasing and live actual parameters). This can be done by analyzing $rev1$ with a reference environment which maps every variable to 1 and ignoring the aliasing of $rev1$ caused by all calls to $rev1$ which do not arise from the body of $rev1$. In terms of the call graph of the program, we reanalyze $rev1$ ignoring the effects of those functions which do not belong to the strongly connected component of $rev1$. We introduce explicit copying for those arguments of a call to $rev1$ which are either live or which cause aliasing of $rev1$ and can potentially be updated in $rev1$. The transformed program is:

```

rev a n      =  rev1(Copy(a), a, 0, n);
rev a b i n  =  if i = n then a
                else
                rev1(Upd(a, i, sel(b, n ⊔ ⊥i)), b, i+1, n);

```

In this program, by introducing explicit copying we have reduced the time and space complexities of the algorithm to $O(n)$.

11 Related Work and Future Research

The earliest work on storage optimization found a linear order of evaluation of the nodes of a labeled dag where the labels represent identifiers, nodes represent assignment statements, and the edges represent data dependencies; it was formalized by Sethi as a pebble game on graphs with labels [21]. Sethi’s work applies to basic blocks with only primitive operators. We assume arbitrary functions as operators,

which necessitates our interprocedural analysis. After deriving the interprocedural information, we derive an order locally in essentially the same way as Sethi.

The other research work in the area of storage optimization is globalization of variables. Schmidt [18] gave the syntactic criteria for converting the store argument of the direct semantics of an imperative language into a global variable. This work was generalized as the globalization of function parameters by Sestoft [20, 11]. It also assumes a fixed order of evaluation of expressions. Fradet [10] gave a simple syntactic criteria, based on the types of the variables, for detecting single threadedness in programs written in continuation passing style.

The initial work on call-by-value functional languages is Hudak’s abstract reference counting technique for a first-order language with flat aggregates [14]. This work assumes a fixed order of evaluation of expressions. We have already given a comparison of our work to this work. Our analysis can be thought of as a generalization of Hudak’s work since we derive an order of evaluation. Our abstraction functions are much simpler than the ones used in [14]. Gopinath [12] considers copy elimination in the single assignment language SAL, which has constructs for specifying *for* loops. His work involves computing the target address of an object returned by an expression using a syntactic index analysis and assuming the liveness analysis of [14]. Again, this work also does not consider reordering expressions. The SAL language has other array creation operators like *cat*, the array concatenation operator, which we haven’t considered in our language.

Bloss [4, 5] extended the work on update analysis to first-order lazy functional languages which are more difficult to analyze because the order of evaluation of expressions cannot be completely determined at compile-time. She defines a non-standard semantics called path semantics [4, 6] which gives the information about all possible orders of the evaluation of variables in a program. Path semantics is used for checking whether an update can be performed destructively. Computing the abstract path semantics is very expensive because of the size of the abstract domain of paths [4]. This work also assumes a fixed order of evaluation of strict operators. Draghicescu’s [9] work on the update analysis of lazy languages improves the abstract complexity but is still exponential.

Deutch [8] describes an analysis based on abstract interpretation, for determining the lifetime and aliasing information for higher-order languages. This analysis is based on the abstraction of the operational semantics of a very low level intermediate language. Our work differs from this work in two ways. Since we do not associate objects to expression labels generating them, we do not introduce spurious aliasing. This work does not address the issue of complexity of the analysis so it is not clear if their work restricted to the first-order case is efficient. Moreover, this work also assumes a fixed order of evaluation of expressions.

James Hicks [13] derives the lifetime information of objects in Id, a parallel single assignment language developed at MIT [16]. Lifetime information is used for validating the deallocate instructions in the program or automatically inserting deallocate instructions for reclaiming the storage. This work does not address the aggregate update problem because Id does not provide update as a language construct (although update operation can be defined in Id).

There are several problems to be considered for future research. One direction is to extend aggregate update analysis for languages with non-flat aggregates. The notion of propa-

gation of an aggregate becomes more complex when non-flat aggregates are allowed in the language. The aliasing analysis described in this paper has to be generalized to a sharing analysis. The other problem is to devise a computable update analysis for higher-order languages. The language we have considered in this paper doesn't have list data structures, so it would be interesting to find suitable abstractions for extending the analysis for languages with list data structures. Another direction to pursue is to find how order of evaluation analysis can be used in lazy functional languages. Bloss's work does not use the strictness information of user defined functions and built-in operators for reordering expression evaluation. It would be worthwhile to study how to derive an order of evaluation for the strict arguments to a function in a lazy functional language, given the strictness information for all the functions in the program.

Acknowledgements

We thank James Hicks and Art Farley for their useful comments and suggestions on an earlier draft of this paper.

References

- [1] S. Abramsky, editor. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd., 1987.
- [2] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms*. Addison Wesley Publishing Company, 1974.
- [3] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools*. Addison Wesley Publishing Company, 1986.
- [4] A. Bloss. *Path Analysis and Optimization of Non-strict Functional Languages*. PhD thesis, Yale University, Dept. of Computer Science, 1989.
- [5] A. Bloss. Update analysis and efficient implementation of functional aggregates. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, 1989.
- [6] A. Bloss and P. Hudak. Path semantics. In *Third Workshop On Mathematical Foundations of Programming Language Semantics*, pages 476–489, 1988.
- [7] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM POPL*, 1977.
- [8] A. Deutch. On determining the lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 1990 POPL*, pages 157–168, 1990.
- [9] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation-order and its applications. In *ACM conference on Lisp and Functional Programming*, pages 242–249, 1990.
- [10] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture*, pages 241–258. ACM, Springer Verlag, August 1991. LNCS 523., 1991.
- [11] C. Gomard and P. Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantic Based Program Manipulation (PEPM)*, pages 166–176, 1991.
- [12] K. Gopinath. *Copy Elimination in Single Assignment Languages*. PhD thesis, Stanford University, Computer Systems Laboratory, 1988.
- [13] J. Hicks Jr. *Compiler Directed Storage Reclamation using Object Lifetime Analysis*. PhD thesis, Electrical Engineering and Computer Science, MIT, 1992.
- [14] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 Conference on Lisp and Functional Programming*, 1986.
- [15] T. Johnsson. Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture, Nancy, France*. Springer Verlag LNCS 523, September 1985.
- [16] R. Nikhil. Id (version 90.0) reference manual. Technical Report CSG Memo 284-1, 545 Technology Square, Cambridge, MA 02139, August 1990.
- [17] A. Sastry and W. Clinger. Order-of-evaluation analysis for destructive updates in strict functional languages with flat-aggregates. Technical report, University of Oregon, 1992. Dept. of Computer Science, TR-92-14.
- [18] D. Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299:310, 1985.
- [19] D. A. Schmidt. *Denotational Semantics : A Methodology for Language Development*. Boston : Allyn and Bacon, 1986.
- [20] P. Sestoft. Replacing function parameters by global variables. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.
- [21] R. Sethi. Pebble games for studying storage sharing. *Theoretical Computer Science*, 19(1):69–84, July 1982.