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

A.V.S. Sastry, William Clinger

CIS-TR-92-14
July 92

Abstract

The aggregate update problem in functional languages is concerned with detecting cases where an 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 an analysis, for strict functional languages with flat aggregates, that derives a good order of evaluation for making the updates destructive. We show that our update analysis algorithm runs in polynomial time. We implemented the algorithm and tested it on some common example programs. The results show that a good choice of the order of evaluation indeed makes most of the updates destructive.

Department of Computer and Information Science
University of Oregon

1 Introduction

The array data structure, with select and update operations associated with it, 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 and 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, thereby degrading the time complexity of an algorithm in proportion to the size of the largest array.

The update operation can be performed in constant time, provided the current copy of the array is no longer live following the update. This constant-time updating is also referred to as destructive or in-place updating. One of the earliest techniques proposed is to use reference counting to perform a dynamic check for the above condition. The first compile-time analysis was proposed by Hudak [Hud86], who described an abstract reference counting technique for a call-by-value first-order language with flat aggregates and a fixed left to right order of evaluation. Subsequently, Bloss [Blo89] considered a first-order lazy functional language with only flat aggregates and a fixed order of evaluation of the strict operators. She defines a nonstandard semantics called path semantics for computing the sequence of formal parameters evaluated in an expression. The information obtained by path semantics is used to decide whether an update can be optimized.

In this paper, we consider the problem of choosing a good order of evaluation for expressions in a call-by-value, first-order functional language having flat aggregates that allows more updates to be implemented destructively in constant time. In a side-effect-free language, the order of evaluation of subexpressions of an expression does not affect the result of the expression. Therefore, if we can evaluate all the expressions that use an aggregate before the expression that updates it, then the update can be made destructively. We propose a static analysis technique to find an order of evaluation and the set of update operators that can be optimized with that order. Our static analysis is based on the technique of abstract interpretation[Abr87] in which a non-standard meaning of a program is computed by choosing an appropriate abstract domain and a proper abstraction of the user defined functions to capture the various properties of interest.

We designed and implemented an abstract interpreter to perform the update analysis. Our results show that for most of the examples, a good order of evaluation makes all the updates destructive, whereas any 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 functional language chosen for analysis. We describe an intermediate language for representing programs and give the transformation from the source language to the intermediate representation. Section 3 illustrates how the order of evaluation is useful for update analysis. Section 4 presents some

definitions and notation used in the later sections. Section 5 describes the abstract domains and abstract semantic functions needed for the analysis. Section 6 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 7. In section 8, we show that our analysis algorithm runs in polynomial time. The results of the implementation of our algorithm are presented in section 9. Section 10 concludes by discussing further work that needs to be done in this area.

2 The Source Language

Our language (figure 1.a) is a first-order, call-by-value language with flat aggregates—an aggregate can contain only non-aggregate values. 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. Our language doesn't allow nested function definitions and free variables. This is not a serious limitation because any first-order functional program with nested definitions can be converted to program with flat definitions by lambda lifting.

2.1 The Intermediate Language

We convert the source program into an intermediate representation using LET-expressions. In this representation of a program, each subexpression which is not a formal parameter or a constant is given a unique name which can be thought of as a compiler generated temporary variable. We say that an expression is *trivial* if it is a formal parameter of a function, or a temporary variable, or a constant. All other expressions can be expressed in terms trivial expressions, if-expressions, LET-expressions, and application of operators and user defined functions to trivial expressions.

The syntax of the intermediate language is shown in figure 1.b. The scope of a LET-binding $t_i = e_i$ in a LET-expression LET [..., $t_i = e_i$, ..., $t_n = e_n$] IN texp END consists of all the occurrences of t_i in expressions e_{i+1} to e_n and texp. An expression in the source language is converted into the intermediate form such that there are no nested LET-expressions. We define a function $\mathcal{T}$ that translates the source program into the intermediate form. In this definition, the symbol @ represents the list append operator. We use ML-style pattern matching to extract the different components of a syntactic object. To extract the LET-bindings of a LET-expression *exp* we write LET $elist_1$ IN t_1 END = *exp* where $elist_1$ is bound to the list of LET-bindings in *exp* and t_1 is the result of the LET-expression. We use a generator called new_temp() for creating new temporary variable names. The definition of $\mathcal{T}$ is given in figure 2. In this translation, we do not have a direct nesting of LET-expressions. The only way a LET expression appears inside another LET-expression is through the conditional expression. Like quadruples, our intermediate

c	$\in$	*Cons*	the set of constants
x	$\in$	V	Variables
op	$\in$	*Prims*	primitive functions like plus, minus etc.
f	$\in$	*FunVar*	user defined functions
exp	$\in$	*Exp* ::=	$c \mid x \mid upd(exp_1, exp_2, exp_3) \mid$
			$sel(exp_1, exp_2) \mid$
			$op(exp_1, \ldots, exp_n) \mid$
			if exp_0 then exp_1 else exp_2
			$f_i(exp_1, \ldots, exp_n)$
pr	$\in$	*Program* ::=	$\{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$

Figure 1.a: The Syntax of Our language

c	$\in$	*Cons*	the set of constants
x	$\in$	V	variables or formal parameters
op	$\in$	*Prims*	primitive functions like plus, minus etc.
f	$\in$	*FunVar*	user defined functions
t_i	$\in$	*TempVars*	names of subexpressions
$texp$	$\in$	*TrivExp* ::=	$c \mid x \mid t_i$
exp	$\in$	*IExp* ::=	$texp \mid upd(texp_1, texp_2, texp_3)$
			$\mid sel(texp_1, texp_2) \mid$
			$op(texp_1, \ldots, texp_n) \mid$
			if $texp$ then exp else $exp \mid$
			$f_i(texp_1, \ldots, texp_n) \mid$
			LET $[t_1 = e_1, \ldots, t_n = e_n]$ IN t_i END
pr	$\in$	*IProg* ::=	$\{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$

Figure 1.b: The Syntax of the Intermediate Language

$$
\begin{array}{lcl}
\mathcal{T} & : & Exp \rightarrow IExp \\
\mathcal{T}[\![\mathrm{c}]\!] & = & \text{LET [] IN } c \text{ END} \\
\mathcal{T}[\![x]\!] & = & \text{LET [] IN } x \text{ END} \\
\mathcal{T}[\![op_i(exp_1,\ldots,exp_n)]\!] & = & \textbf{let LET } list_1 \text{ IN } t_1 \text{ END } = \mathcal{T}[\![exp_1]\!] \\
 & & \quad \vdots \\
 & & \quad \text{LET } list_n \text{ IN } t_n \text{ END } = \mathcal{T}[\![exp_n]\!] \\
 & & \quad t = new\_temp() \\
 & & \textbf{in} \\
 & & \quad \text{LET } list_1@\ldots@list_n@[t = op(t_1,\ldots,t_n)] \text{ IN } t \text{ END} \\
 & & \textbf{end} \\
\mathcal{T}[\![\mathrm{sel}(exp_1,exp_2)]\!] & = & \textbf{let LET } list_1 \text{ IN } t_1 \text{ END } = \mathcal{T}[\![exp_1]\!] \\
 & & \quad \text{LET } list_2 \text{ IN } t_2 \text{ END } = \mathcal{T}[\![exp_2]\!] \\
 & & \quad t = new\_temp() \\
 & & \textbf{in} \\
 & & \quad \text{LET } list_1@list_2@[t = sel(t_1,t_2)] \text{ IN } t \text{ END} \\
 & & \textbf{end} \\
\mathcal{T}[\![upd_i(exp_1,exp_2,\ exp_3)]\!] & = & \textbf{let LET } list_1 \text{ IN } t_1 \text{ END } = \mathcal{T}[\![exp_1]\!] \\
 & & \quad \text{LET } list_2 \text{ IN } t_2 \text{ END } = \mathcal{T}[\![exp_2]\!] \\
 & & \quad \text{LET } list_3 \text{ IN } t_3 \text{ END } = \mathcal{T}[\![exp_3]\!] \\
 & & \quad t = new\_temp() \\
 & & \textbf{in} \\
 & & \quad \text{LET } list_1@list_2@[t = upd(t_1,t_2,t_3)] \text{ IN } t \text{ END} \\
 & & \text{end} \\
\mathcal{T}[\![\text{if } exp_0 \text{ then } exp_1 \text{ else } exp_2]\!] & = & \textbf{let LET } list_0 \text{ IN } t_0 \text{ END } = \mathcal{T}[\![exp_0]\!] \\
 & & \quad t = new\_temp() \\
 & & \textbf{in} \\
 & & \quad \text{LET } list_0@[t = \text{if } t_0 \text{ then } \mathcal{T}[\![exp_1]\!] \text{ else } \mathcal{T}[\![exp_2]\!]] \\
 & & \quad \text{IN } t \text{ END} \\
 & & \text{end} \\
\mathcal{T}[\![f_i(exp_1,\ldots,exp_n)]\!] & = & \textbf{let LET } list_1 \text{ IN } t_1 \text{ END } = \mathcal{T}[\![exp_1]\!] \\
 & & \quad \vdots \\
 & & \quad \text{LET } list_n \text{ IN } t_n \text{ END } = \mathcal{T}[\![exp_n]\!] \\
 & & \quad t = new\_temp() \\
 & & \textbf{in} \\
 & & \quad \text{LET } list_1@\ldots@list_n@[t = f_i(t_1,\ldots,t_n)] \text{ IN } t \text{ END} \\
 & & \text{end}
\end{array}
$$

Figure 2: The Translation Function $\mathcal{T}$

representation is convenient for finding and expressing a good order of evaluation. After the analysis, all the LET-bindings are reordered. The reordered program is subsequently used in the abstract reference count analysis and the code generation phases of the compiler. We make the following assumption about the temporary names created. The names of the temporary variables are linearly ordered such that if expression e_i is a subexpression of e_j, then i is less than j. This fact is used subsequently in deriving an order of evaluation consistent with the dependence relation among the expressions.

3 An Example

We motivate the need for deriving an order of evaluation through a simple example. Consider the following program

$$\begin{aligned} f\ x\ i &= \text{if } i = 0 \text{ then } x \text{ else upd}(x,i,2i) \\ g\ y\ j &= sel(y,j) + sel(f(y,j),j) + sel(y,2j) \end{aligned}$$

The function f updates its first argument conditionally depending on the value of i. We observe that any fixed order of evaluation of the binary + operator cannot make the update destructive. For example, if we assume left to right evaluation of the +, then destructive updating would change the value of y, thereby interfering with evaluation of the third select operator. In a right to left evaluation, destructive updating would affect the result returned by the first select operator. For this example, the only way to make the update destructive is to evaluate the first occurrence of + from left to right and the second occurrence from right to left. The aim of our analysis is to derive such an order of evaluation for each subexpression of the program with the objective of making the updates destructive.

The above example may lead the reader to believe that perhaps a suitable order of evaluation can make all the updates destructive. But we can easily come up with examples where no order of evaluation can make all the updates destructive. Consider another program

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

In this example, there is no order of evaluation of + that can make the update destructive. In such cases, we choose some arbitrary order of evaluation and analyze the program with that chosen order. Even for the cases where no order makes all the updates destructive, one order may be better than another. For example, a particular order may optimize more lexical occurrences of updates. Another order may optimize an update that is executed more frequently. Ideally

speaking, we would like to optimize the updates that are executed most often. Since we cannot determine at compile time how many times an update may be executed, we have to apply heuristics to decide which update to optimize. An update occurring in a recursive call is likely to be executed more often. In such cases the choice of an order must necessarily be heuristic. In this paper we do not discuss this issue any further.

4 Definitions and Notation

Now we introduce some definitions and notation useful for the discussion subsequently. Multiple references to an aggregate are created if it is bound to a formal parameter that occurs multiple times in the body of a function. Multiple references can also be created by the aliasing of formal parameters.

4.1 Definitions

An expression is said to *propagate* a variable if one of the possible results of evaluation of the expression is the variable. If an aggregate is bound to a variable in an expression, and if that variable is propagated by the expression, then we say that the aggregate is propagated by the expression.

An expression is said to *consume* all the variables appearing in it if it does not propagate any of its variables. If an expression consumes all the variables appearing in it, all the references that occur in the expression become dead after the expression evaluation.

An expression e_j is *dependent* on e_i if e_i is a subexpression of e_j. The dependency relation between subexpressions of an expression can be represented as a directed graph which we call the *dependency* graph. In this graph, each node represents a subexpression. Each edge is a dependency between two subexpressions.

Two formal parameters of a function are said to be *aliased* to each other if there exists a function call in which the formal parameters can potentially be bound to the same object (the same aggregate). The aliasing information is important for the update analysis. Consider a program

$$\begin{aligned} f\,x\,i &= \text{if } i = 0 \text{ then } x \text{ else } upd(x,i,i) \\ g\,x\,y\,j &= sel(f(y,j),j+1) + sel(x,j) \\ h\,x\,i &= g(x,x,i) \end{aligned}$$

In the body of function g, even though x and y appear to be distinct, they are aliased because of the call to g in the function h. A global analysis called *aliasing analysis* finds out the possible aliases of formal parameters of a function.

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 in not permissible because it would change the semantics of the expression e_1. The notion of interference is used subsequently to augment the dependence graph associated with a LET-expression with additional edges called *interference-edges* and the resulting graph is used to derive an order of evaluation of expressions.

4.2 Notation

We assume that the variables in the program are renamed so that each variable is distinct. We use the notation $x_{k:j}$ to refer to the jth formal parameter of the function f_k in the program. We assume that the reader is familiar with partial orders, the least upper bound (lub) operation which is denoted by $\bigsqcup$, 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.

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, ..., x_n to v_n is written as $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. 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

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

5 The Abstract Functions

The denotational semantics of a language gives the meaning of a program in terms of the meanings of its parts. It is a function which maps elements of the syntactic domains in a given environment (a map from identifiers to values of the domain) to the elements of the semantic domains. An environment is used for specifying the meaning of the free identifiers. The meaning of a program is specified by specifying a domain and a semantic function. The domain which represents the actual values of computations is called the standard domain.

Abstract interpretation can be viewed as defining the meaning of a program using a non-standard domain called the abstract domain. The elements of the abstract domain represent properties of the elements of the standard or concrete domains. The functions on the actual domain become abstract functions on the abstract domain. Since the abstract domain is simpler and smaller than the actual domain, the computations in the abstract domain are also less

complex and can be made to terminate even if the actual computations do not terminate. The purpose of abstract interpretation is to extract properties of programs at compile time by defining the appropriate abstract domain and abstract functions and performing the abstract computations. In this section, we define three abstract domains and abstraction functions which are used for update analysis. Each abstraction gives one particular meaning to the program which captures one particular property of interest. The three properties we need are the set of variables propagated by an expression, the aliasing of the variables of the program, and the set of aggregates selected and updated in an expression evaluation. The information computed can be used for constructing the interference graph and finding a good ordering of subexpressions. Since all the domains are finite and the functions are monotonic, the fixpoints can be computed in finite time.

All the abstract functions are defined on the syntactic objects of the intermediate language. For the purposes of the definitions of these functions, the formal parameters and the temporary variables are treated identically. 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 define three different meanings $\mathcal{H}_p[\![\text{pr}]\!]$, $\mathcal{A}_p[\![\text{pr}]\!]$, and $\mathcal{S}_p[\![\text{pr}]\!]$ of a program pr.

5.1 Propagation Analysis

Given an expression containing variables, we need to determine the set of variables that can be propagated by the expression. The set of variables are the formal parameters of the user defined functions.

The abstract domains needed for this analysis are shown in figure 3.a. The abstraction function $\mathcal{H}$ (see figure 3.b) computes the set of variables propagated by an expression in a given variable and function environments.

The set of variables propagated by a constant is empty. The set of variables propagated by a variable is obtained by looking up the variable in the given environment. The set of variables propagated by a primitive expression [1] depends on the semantics of primitive operators. Since primitive operators of our language do not propagate any of their arguments, the set of variables propagated by the primitive expressions is also empty. As our language does not permit non-flat aggregates, a *sel* 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. In the case of an if-expression, the sets of variables propagated by both the branches are computed recursively and the union of these sets is returned as the result. For a function call, the sets of variables propagated by each actual parameter is computed recursively.

[1] an expression obtained by applying a primitive operator to trivial expressions

V			program variables (formal parameters of functions)
F			user defined functions
D	$=$	$\mathcal{P}(V)$	the powerset domain with set union as the lub operation on D
$VEnv$	$=$	$V \rightarrow D$	the domain of abstract variable environments
$FEnv$	$=$	$F \rightarrow D^n \rightarrow D$	the domain of abstract function environments

Figure 3.a: The Abstract Domains for Propagation Analysis

$$
\begin{array}{lll}
\mathcal{H} : IExp \rightarrow VEnv \rightarrow FEnv \rightarrow D & & \\
\mathcal{H}[\![c]\!]\sigma\,\rho & = & \emptyset \\
\mathcal{H}[\![x]\!]\sigma\,\rho & = & \sigma[x] \\
\mathcal{H}[\![op_i(exp_1,\ldots,exp_n)]\!]\sigma\,\rho & = & \emptyset \\
\mathcal{H}[\![\mathrm{sel}(exp_1,exp_2)]\!]\sigma\,\rho & = & \emptyset \\
\mathcal{H}[\![upd_i(exp_1,exp_2,exp_3)]\!]\sigma\,\rho & = & \emptyset \\
\mathcal{H}[\![\text{if } exp_0 \text{ then } exp_1 \text{ else } exp_2\,]\!]\sigma\,\rho & = & \mathcal{H}[\![exp_1]\!]\sigma\,\rho \;\cup\; \mathcal{H}[\![exp_2]\!]\sigma\,\rho \\
\mathcal{H}[\![f_k(exp_1,\ldots,exp_n)]\!]\sigma\rho & = & \rho[f_k](\mathcal{H}[\![exp_1]\!]\sigma\rho,\ldots,\mathcal{H}[\![exp_n]\!]\sigma\,\rho) \\
\mathcal{H}[\![\text{LET } [t_1 = exp_1,\ldots,t_n = exp_n] \text{ IN texp END}]\!]\sigma\,\rho & = & \\
\quad \textbf{let } v_1 = \mathcal{H}[\![exp_1]\!]\sigma\,\rho & & \\
\qquad v_2 = \mathcal{H}[\![exp_2]\!]\sigma[t_1 \mapsto v_1]\,\rho & & \\
\qquad \vdots & & \\
\qquad v_n = \mathcal{H}[\![exp_n]\!]\sigma[t_1 \mapsto v_1,\ldots,t_{n-1} \mapsto v_{n-1}]\,\rho & & \\
\quad \textbf{in} & & \\
\qquad \mathcal{H}[\![texp]\!]\sigma[t_1 \mapsto v_1,\ldots,t_n \mapsto v_n]\,\rho & & \\
\quad \textbf{end} & &
\end{array}
$$

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

$$
\begin{array}{lll}
\mathcal{H}_p & : & IProg \rightarrow FEnv \\
\mathcal{H}_p[\![pr]\!] & = & fix(\lambda\rho.\,\rho[f_i \mapsto \lambda y_1,\ldots,y_n.\mathcal{H}[\![exp_i]\!][x_{i:1} \mapsto y_1,\ldots,x_{i:n} \mapsto y_n]\,\rho])
\end{array}
$$

Figure 3.c: The function $\mathcal{H}_p$ for computing $\mathcal{H}$-meaning of a program

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 computes the $\mathcal{H}$-meaning of a program. Given a program $pr = \{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$, $\mathcal{H}_p$ 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. 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 abstract propagation environment.

5.2 Aliasing Analysis

For a safe update analysis, we need to determine the aliasing information about a program. Two formal parameters of a function are aliased if there exists a function call in which the two formal parameters are bound to same aggregate. We determine if two formal parameters can potentially be aliased in any program execution. The aliasing of parameters of a function can cause aliasing in other functions. 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. We define the abstract function $\mathcal{A}$ (see figure 4.b) that computes the aliasing information in a program. The semantic function $\mathcal{A}$ uses the abstract propagation environment and the function $\mathcal{H}$ in its definition. The function $\mathcal{A}$ takes an expression, an element of the aliasing 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. For each pair of formal parameters of the function, we check 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 potentially be aliased. We update the aliasing information of the parameters of the called function. If the formal parameters $x_{k:i}$ and $x_{k:j}$ could be aliased then the aliases of $x_{k:i}$ and $x_{k:j}$ have to be updated with new aliases. The new alias-set is the union of aliases of $x_{k:i}$ and $x_{k:j}$. The new aliasing environment maps each alias of $x_{k:i}$ and $x_{k:j}$ to the new alias-set. All other cases of the definition of function $\mathcal{A}$ are straightforward.

The $\mathcal{A}$-meaning of a program $pr = \{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$ is computed by $\mathcal{A}_p$ which takes a program as argument and returns the aliasing information of all the user-defined functions in the program. The aliasing environment of a program is computed by the fixpoint equation shown in the figure 4.c. In this equation, σ_{id} is the identity environment in which every variable is bound to a singleton set containing itself. The aliasing environment is

$D = \mathcal{P}(V)$ the power set of variables with set union as the lub operation.
$AEnv = V \rightarrow D$ the domain of aliasing environments

Figure 4.a: The Abstract Domains for Aliasing Analysis

$$
\begin{array}{l}
\mathcal{A} : IExp \rightarrow AEnv \rightarrow FEnv \rightarrow AEnv \\
\mathcal{A}[\![\mathrm{c}]\!]\sigma\,\rho = \sigma \\
\mathcal{A}[\![x]\!]\sigma\,\rho = \sigma \\
\mathcal{A}[\![op_i(exp_1,\ldots,exp_n)]\!]\sigma\,\rho = \sigma \\
\mathcal{A}[\![\mathrm{sel}(exp_1,exp_2)]\!]\sigma\,\rho = \sigma \\
\mathcal{A}[\![upd_i(exp_1,exp_2,exp_3)]\!]\sigma\,\rho = \sigma \\
\mathcal{A}[\![\text{if } exp_0 \text{ then } exp_1 \text{ else } exp_2\ ]\!]\sigma\,\rho = \mathcal{A}[\![exp_1]\!]\sigma\,\rho \sqcup \mathcal{A}[\![exp_2]\!]\sigma\,\rho \\
\mathcal{A}[\![f_k(exp_1,\ldots,exp_n)]\!]\sigma\rho = \\
\quad \textbf{let} \\
\qquad v_i = \mathcal{H}[\![exp_i]\!]\sigma\,\rho \\
\qquad a_{ij} = v_i \cap v_j, 1 \le i,j \le n, i \ne j \\
\quad \textbf{in} \\
\qquad \sigma \sqcup [x_{k:l} \mapsto (\sigma[x_{k:i}] \cup \sigma[x_{k:j}]) \mid a_{ij} \ne \emptyset, x_{k:l} \in (\sigma[x_{k:i}] \cup \sigma[x_{k:j}])] \\
\quad \textbf{end} \\
\mathcal{A}[\![\text{LET } [t_1 = exp_1,\ldots,t_n = exp_n] \text{ IN texp END}]\!]\sigma\,\rho = \bigsqcup_{i=1}^{n} \mathcal{A}[\![exp_i]\!]\sigma\,\rho
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{A}_p : IProg \rightarrow AEnv \\
\mathcal{A}_p[\![pr]\!] = \\
\quad \textbf{let } \rho = \mathcal{H}_p[\![pr]\!] \\
\quad \textbf{in} \\
\qquad fix(\lambda\sigma.\ \bigsqcup_{i=1}^{n} \mathcal{A}[\![exp_i]\!]\sigma\,\rho \sqcup \sigma_{id}) \\
\quad \textbf{end}
\end{array}
$$

Figure 4.c: The function $\mathcal{A}_p$ for computing the $\mathcal{A}$-meaning of a program

used for determining the liveness of a variable in an expression. We say that a variable is *live* at a program point if there is subsequent use of the variable or its alias in the rest of the program.

5.3 Selects-and-Updates Analysis

Destructive updating of an aggregate in an expression evaluation can affect the evaluation of other expressions which use the old aggregate. To determine how evaluation of an expression with destructive updating may affect another expression, we need to determine the sets of aggregates that are possibly selected and possibly updated in an expression evaluation. To compute these sets, we use a product domain as the abstract domain. The first component of an element of the abstract domain 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 domains needed for the selects-and-updates analysis are given in figure 5.a. We use the abstract propagation environment computed by the function $\mathcal{H}$, and the function $\mathcal{H}$ for determining the set of aggregates selected and updated by an expression. 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 arguments.

The semantic function $\mathcal{S}$ computes the two sets of aggregates selected and updated when a particular expression is evaluated. The lub operation on the product domain D_{su} is the componentwise set union. The definition of $\mathcal{S}$ is given in figure 5.b.

For a constant, or a variable, or a primitive expression, no aggregate is selected or updated. For a select expression, the set of aggregates selected is the set of variables propagated by its first argument. For an update expression, the set of aggregates updated is the set of variables propagated by its first argument. The sets of aggregates selected and updated by a function call are determined by applying the corresponding abstract function to the variables propagated by each of the arguments to the function. In the case of a LET-expression, the sets of aggregates selected and updated by all the LET-bindings are computed their lub is returned. Similarly, for an if-expression the sets of aggregates selected and updated by both the branches are computed and their lub is returned as the result.

The $\mathcal{S}$-meaning of a program is an environment that maps each function symbol to an abstract function which gives the set of variables selected and updated, given the set of variables bound to each parameter of the function. The function $\mathcal{S}_p$ which computes the $\mathcal{S}$-meaning of a program $pr = \{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$, is defined as the fixpoint of an equation as shown in figure 5.c. The information computed by the function $\mathcal{S}$ is used in deriving an order of evaluation of expressions.

$$
\begin{array}{lcll}
D_{su} & = & \mathcal{P}(V) \times \mathcal{P}(V) & \text{possibly selected} \times \text{possibly updated} \\
VEnv & = & V \to \mathcal{P}(V) & \text{the domain of abstract variable environments} \\
FEnv & = & F \to \mathcal{P}(V)^n \to \mathcal{P}(V) & \text{the domain of abstract function environments} \\
SEnv & = & F \to \mathcal{P}(V)^n \to D_{su} & \text{the domain of selects-and-updates environment}
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{S} : IExp \to SEnv \to FEnv \to VEnv \to D_{su} & & \\
\mathcal{S}[\![\text{c}]\!]\sigma\ \delta\ \rho & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![x]\!]\sigma\ \delta\ \rho & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![op_i(exp_1, \ldots, exp_n)]\!]\sigma\ \delta\ \rho & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![\text{sel}(exp_1, exp_2)]\!]\sigma\ \delta\ \rho & = & \langle \mathcal{H}[\![exp_1]\!]\delta\ \rho, \emptyset \rangle \\
\mathcal{S}[\![upd_i(exp_1, exp_2, exp_3)]\!]\sigma\ \delta\ \rho & = & \langle \emptyset, \mathcal{H}[\![exp_1]\!]\delta\ \rho \rangle \\
\mathcal{S}[\![\text{if } exp_0 \text{ then } exp_1 \text{ else } exp_2\ ]\!]\sigma\ \delta\ \rho & = & \mathcal{S}[\![exp_1]\!]\sigma\ \delta\ \rho\ \sqcup\ \mathcal{S}[\![exp_2]\!]\sigma\ \delta\ \rho \\
\mathcal{S}[\![f_k(exp_1, \ldots, exp_n)]\!]\sigma\ \delta\ \rho & = & \textbf{let} \\
 & & \quad v_1 = \mathcal{H}[\![exp_1]\!]\delta\ \rho \\
 & & \quad \vdots \\
 & & \quad v_n = \mathcal{H}[\![exp_n]\!]\delta\ \rho \\
 & & \textbf{in} \\
 & & \quad \sigma[f_k](v_1, \ldots, v_n) \\
 & & \textbf{end} \\
\mathcal{S}[\![\text{LET } [t_1 = exp_1, \ldots, t_n = exp_n] \text{ IN texp END}]\!]\sigma\ \delta\ \rho & = & \bigsqcup_{i=1}^{n} \mathcal{S}[\![exp_i]\!]\sigma\ \delta\ \rho
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{S}_p & : & IProg \to SEnv \\
\mathcal{S}_p[\![pr]\!] & = & fix(\lambda\sigma.\ \sigma[f_k \mapsto \lambda x_1, \ldots, x_n.\ \mathcal{S}[\![exp_k]\!]\sigma\ \ (\mathcal{H}_p[\![pr]\!])\ \ [x_{k:1} \mapsto x_1, \ldots, x_{k:n} \mapsto x_n]])
\end{array}
$$

Figure 5.c: The function $\mathcal{S}_p$ for computing the $\mathcal{S}$-meaning of a program

6 Deriving an Order of Evaluation

In a call-by-value pure functional language, the arguments of a function are evaluated before the function is called. Semantically, the order of evaluation of the arguments is immaterial to the result computed by the function. Therefore, the compiler has the freedom to choose any order of evaluation for the arguments of a function call. Since every expression of the source language is converted into a LET-expression in the intermediate language, we need to derive an order of evaluation for the LET-bindings of a LET-expression. Our objective is to choose an order of evaluation that allows us to perform the updates destructively.

Consider a let expression LET $[t_1 = e_1, \ldots, t_n = e_n]$ IN texp END. To find the order of evaluation of expressions t_1 to t_n , we construct a directed graph. We start with the dependence graph whose nodes are the expressions $t_1, \ldots, t_n$. Recall that we say an expression t_j depends on t_i if t_i is a subexpression of t_j (or t_i appears in t_j). In the dependence graph, this dependence is represented as directed edge (i,j), indicating that the node i must be evaluated before the node j. The dependence graph will necessarily be a directed acyclic graph (dag) because we are assuming strict semantics and our language doesn't allow cyclic data structures. The dependence graph can be constructed in time linear in the number of nodes. The compiler has the freedom to choose a suitable ordering for only those expressions which are not related to one another by the dependency relationship. To perform the updates destructively, we have to make sure that all the selects on an aggregate are performed before the update. A suitable ordering is obtained by forcing the evaluation of those expressions which select an aggregate before the evaluation of expressions that update the aggregate. This is done by augmenting the dependence graph with additional edges which represent the desired dependencies. We call these additional edges as the *interference-edges* . An interference edge (i,j) conveys the information that e_j possibly updates an aggregate needed by e_i. The notion of interference is defined in terms of the aggregates selected and updated by the two expressions. Given two expressions e_i and e_j, evaluation of e_j interferes with the evaluation of e_i, because of destructive updating, if e_j updates some aggregate that is also possibly selected or updated by e_i. Formally, interference is defined as

$$
\begin{aligned}
\mathit{interferes}\ e_i\ e_j \quad = \quad & \textbf{let} \\
& \quad < s_i, u_i > = \mathcal{S}[\![e_i]\!] \mathit{suenv}\ \mathit{fenv}\ \mathit{aenv} \\
& \quad < s_j, u_j > = \mathcal{S}[\![e_j]\!] \mathit{suenv}\ \mathit{fenv}\ \mathit{aenv} \\
& \textbf{in} \\
& \quad u_j \cap (s_i \cup u_i) \neq \emptyset \\
& \textbf{end}
\end{aligned}
$$

In the above definition, *suenv* is the $\mathcal{S}$-meaning of the program computed by $\mathcal{S}$, *aenv* is the aliasing information of the program, and *fenv* is the abstract propagation environment, the

$\mathcal{H}$-meaning computed by $\mathcal{H}$. The set of interference edges E is defined as,

$$E = \{(i,j) | interferes(e_i, e_j), 1 \leq i,j \leq n, i \neq j\}$$

We obtain the precedence graph, which gives a partial order for expression evaluation, by adding these interference edges to the dependence graph of the LET-expression. The evaluation of expression e_i should occur before the evaluation of the node e_j if there is a path from node i to j in the precedence graph. There can be cycles in the precedence graph because of addition of interference edges. We find the strongly connected components of the precedence graph using the algorithm given in [AHU74]. 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. It represents the precedences of the evaluation of the strongly the evaluation of the components which subsumes the precedence given by the dag. This linear ordering is obtained by taking a topological sort of the new dag. An ordering of all the expressions is obtained by replacing each component in the topologically sorted component list by some ordering of the elements of the component. In any component, if there are dependence edges as well as interference edges among the nodes, we have to choose an ordering which is a linear extension of the dependence relation. For the components in which all paths only consist of interference edges, any ordering is permissible. To handle both these cases identically, we use the following fact about the ordering of the names of the expressions created during the translation to intermediate form. If a node e_i is dependent on node e_j then i is greater than j. Therefore a suitable ordering of nodes in a component is obtained by sorting the elements of the component in ascending order.

The complexity of deriving the order of evaluation information is $O(n^2)$ where n is the number of expressions in a LET-expression. This bound cannot be improved because we have to consider every two expressions for possible interference. This does not take into account the time taken for finding the abstract semantic functions. We discuss the complexity of these functions in section 8. The dependence graph can be constructed in linear time in the number of edges which is again bounded by $O(n^2)$. For adding the interference edges, we have to consider each pair of expressions which needs $O(n^2)$ time. Strongly connected components, topological sorting of components, and the sorting of elements in all the components can be computed in $O(n^2)$ time.

In an ordered LET-expression, obtained by our ordering algorithm, LET $[t_i = e_i, \ldots, t_k = e_k]$ IN texp END, the expression t_i is evaluated before t_k. Given the order of expression evaluation, one can determine the set of live variables at each LET-binding. The liveness information is used for obtaining an abstraction of the reference count of the objects bound to the formal parameters of each function. The abstract reference count analysis is described in the next section.

7 Abstract Reference Count Analysis

To know whether an aggregate can be updated destructively, we need to know the number of references to it. If we know that there is exactly one reference to the aggregate which is held by the update operator, then the update can be performed destructively. For the analysis, it is sufficient to know if there is one or more than one live reference to an aggregate. Our abstraction of the reference count is a 2-point domain R whose least element 1 represents the existence of exactly one live reference to an aggregate and $\top$ represents the existence of multiple live references.

The abstract reference environment is the abstraction of reference count of objects bound to the formal parameters of a function in all possible calls that could arise in any program execution. For computing the abstract reference count environment, we make use of the aliasing environment, abstract propagation environment, and the set of live variables at each program point. A LET-binding is a program point for the purposes of analysis. Our analysis differs from Hudak's abstract reference count analysis[Hud86] as we do not mimic the operational semantics of the language by allocating locations for variables, initializing reference counts to the number of occurrences of the variables in a function body, and appropriate incrementing and decrementing of reference counts. In our analysis, the live variable set at each program point, the aliasing information, and the function $\mathcal{H}$ are used in computing the liveness of an object at any program point. We take a simplistic view that a variable is live if it occurs in an expression which is evaluated subsequently - we assume that there are no useless variables in the program. The function *Vars* returns the set of variables in an expression. It can be defined recursively on the syntax of the intermediate language.

A variable, representing the object bound to it, can be live within the body of a function in two possible ways. In one case, the actual parameter that was bound to the variable was live when the function was called in some context. In the other case, multiple occurrences of the variable in the function body cause the variable to be live at different program points. We refer to the first type of liveness as *global* liveness and the second type as *local* liveness. A variable is live at a program point if it is either globally or locally live.

The global liveness information is available from the abstract reference environment. The local liveness information can be extracted from the set of live variables. The aliasing information is used for capturing the local liveness of a variable correctly. A variable is live locally at a program point if it is one of the live variables or one of its aliases is a live variable.

The abstract domains and the abstract semantic function $\mathcal{R}$ for computing the abstract reference environment are shown in figures 6.a and 6.b respectively. The function $\mathcal{R}$ updates the reference environment only in the case of a function call. Given a function call, we determine the set of variables propagated by each actual parameter of the function. The liveness of each parameter can be tested by checking if it is globally or locally live. Global liveness of an actual parameter is tested by determining if at least one of the variables propagated by the actual

$$
\begin{array}{lcll}
FEnv & = & F \to \mathcal{P}(V)^n \to \mathcal{P}(V) & \\
AEnv & = & V \to \mathcal{P}(V) & \\
R & = & \{1, \top\} & \text{the domain of abstract reference counts} \\
REnv & = & V \to R & \text{the domain of abstract reference environments} \\
LSet & = & \mathcal{P}(V) & \text{the domain of live variables}
\end{array}
$$

Figure 6.a: The Abstract Domains for Reference Count Analysis

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

$$
\begin{array}{lcl}
\mathcal{R}[\![\text{c}]\!] renv\ fenv\ aenv\ lset & = & renv \\
\mathcal{R}[\![x]\!] renv\ fenv\ aenv\ lset & = & renv \\
\mathcal{R}[\![\text{op}(exp_1, \ldots, exp_n)]\!] renv\ fenv\ aenv\ lset & = & renv \\
\mathcal{R}[\![\text{sel}(exp_1, exp_2)]\!] renv\ fenv\ aenv\ lset & = & renv \\
\mathcal{R}[\![upd_i(exp_1, exp_2, exp_3)]\!] renv\ fenv\ aenv\ lset & = & renv \\
\mathcal{R}[\![\text{if } exp_0 \text{ then } exp_1 \text{ else } exp_2\ ]\!] renv\ fenv\ aenv\ lset & = & \\
\quad \mathcal{R}[\![exp_1]\!] renv\ fenv\ aenv\ lset \sqcup \mathcal{R}[\![exp_2]\!] renv\ fenv\ aenv\ lset & & \\
\mathcal{R}[\![f_k(exp_1, \ldots, exp_n)]\!] renv\ fenv\ aenv\ lset & = & \\
\quad \textbf{let } v_1 = \mathcal{H}[\![exp_1]\!] fenv\ aenv & & \\
\qquad \vdots & & \\
\qquad v_n = \mathcal{H}[\![exp_n]\!] fenv\ aenv & & \\
\quad \textbf{in} & & \\
\qquad renv \sqcup (\sqcup\{[x_{k:j} \mapsto \top\,] \mid \exists x \in v_j, renv[x] = \top \quad \vee \quad x \in lset\}) & & \\
\quad \textbf{end} & & \\
\mathcal{R}[\![\text{LET } [t_1 = e_1, \ldots, t_n = e_n] \text{ IN texp END}]\!] renv\ fenv\ aenv\ lset & = & \\
\quad \textbf{let } lset_i = lset \cup (\cup\{aenv[x] \mid x \in \bigcup_{j=i+1}^{n} Vars(e_j)\}) & & \\
\qquad \cup (\cup\{\mathcal{H}[\![t_k]\!] fenv\ aenv \mid k < i, \exists j,\ i < j < n+1, t_j \text{ is a subexpression of } t_k\}) & & \\
\quad \textbf{in} & & \\
\qquad \bigsqcup_{i=1}^{n} \mathcal{R}[\![e_i]\!] renv\ fenv\ aenv\ lset_i & & \\
\quad \textbf{end} & &
\end{array}
$$

Figure 6.b: The function $\mathcal{R}$

$$
\begin{array}{lll}
\mathcal{R}_p & : & IProg \rightarrow REnv \\
\mathcal{R}_p[\![pr]\!] & = & \\
 & \textbf{let} & fenv = \mathcal{H}_p[\![pr]\!] \\
 & & aenv = \mathcal{A}_p[\![pr]\!] \\
 & \textbf{in} & \\
 & & fix(\lambda\sigma.\bigsqcup_{i=1}^{n}\mathcal{R}[\![exp_i]\!]\sigma\ fenv\ aenv\ \emptyset) \\
 & \textbf{end} &
\end{array}
$$

Figure 6.c: The function $\mathcal{R}_p$ for computing the $\mathcal{R}$-meaning of a program

parameter has the value $\top$ in the reference environment *renv*. Local liveness is checked by finding if at least one of the variables (or its aliases) propagated by the actual parameter is a live variable. In the case of a LET-expression, the set of live variables at each LET-binding is computed and the LET-bindings are analyzed recursively. The live variables at a LET-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 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 $pr = \{f_1\ x_1\ \ldots\ x_k{=}exp_1;\ \ldots;\ f_n\ x_1\ \ldots\ x_k{=}exp_n\}$, the definition of $\mathcal{R}_p$ is given in figure 6.c as the solution to a fixpoint equation.

Given ($\mathcal{R}_p$[pr]), it is easy to decide if an update can be performed destructively. Consider an update expression whose first argument is exp_1. Suppose that *lset* is the set of variables that are live at exp_1. This update cannot be made destructive if there is at least one variable propagated by exp_1 which is either globally live or locally live. This condition is formally expressed as

$(\ \exists x\ \in \mathcal{H}[\![exp_1]\!]fenv\ aenv,\ renv[x] = \top\ )\ \vee\ ((\mathcal{H}[\![exp_1]\!]fenv\ aenv) \cap lset \neq \emptyset)$

In this expression, *fenv*, *aenv*, and *renv* are $\mathcal{H}_p$[pr], $\mathcal{A}_p$[pr], and $\mathcal{R}_p$[pr] of the program pr defined by functions $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{R}$ respectively.

8 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 $m \geq n$ the number of non LET-

expressions in the program. 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 complexity of each iteration. The basic unit of analysis for each of the abstract functions is a non LET-expression. A LET-expression is analyzed by analyzing its LET-bindings. The work in any iteration is bounded by the complexity of analyzing a non-LET expression and multiplying it by the number of non-LET expressions in the program.

8.1 Complexity of $\mathcal{H}_p$

Given a program containing n functions, $\mathcal{H}_p$ computes an environment in which each function symbol is mapped to an abstract function of at most k variables. The abstract propagation function can only be a union of some of its formal parameters : refer to the definitions in figures 3.b and 3.c. For example, the function $\lambda x.\ \lambda y.\ \lambda z.$ x propagates its first argument, $\lambda x.\ \lambda y.\ \lambda z.$ x $\cup$ y propagates its first and second arguments and $\lambda x.\ \lambda y.\ \lambda z.$ x $\cup$ y $\cup$ z propagates all arguments. Any chain of abstraction functions can have length at most k. In the fixpoint computation, we start with the least element of the domain - an environment mapping every function symbol to a function which doesn't propagate any of its arguments. The number of iterations for reaching the fixpoint is bounded by the number of iterations required to reach the top element of the domain $FEnv$. Since each function symbol can take only k different values and the value of at least one function symbol has to change in each iteration, the maximum number of iterations needed is kn, and therefore is the bound on the number of iterations needed for computing $\mathcal{H}_p$.

We can now estimate the complexity of each iteration. This estimate depends on how an abstract function is represented. Since any abstract function is a union of some of its formal parameters, we can represent it by an ordered set of those formal parameters that are propagated by the function body. The maximum size of such a set is k. The most complex operation in the definition of $\mathcal{H}$ (see figure 3.b) is the abstraction of a function call in which the variables propagated by the arguments of the function are computed and the abstraction function corresponding to the called function is applied to these arguments. Each of the arguments can propagate at most k variables, and the abstraction function can at most be the union of k variables which means taking at most k unions of ordered sets each containing at most k elements. The time complexity of an ordered set union is $O(k)$, therefore it takes $O(k^2)$ for the computing the abstraction of a function call. Since the program contains m non LET-expressions, the time for one iteration is $O(mk^2)$.

The time complexity of $\mathcal{H}_p$ is $O(mnk^3)$ which follows from the bound on the number of iterations and the complexity of each iteration.

8.2 Complexity of $\mathcal{A}_p$

Given a program pr, $\mathcal{A}_p$ computes the aliasing environment. If each function has at most k variables, the number of variables in an aliasing environment is $O(nk)$. Each variable can be aliased to at most k variables because we are considering the aliasing among the formal parameters of a function. The number of iterations needed for reaching the top element of the aliasing environment starting with the least element and assuming that at least one variable changes its value in each iteration, is $O(nk^2)$. It is a bound on the number of iterations in the fixpoint calculation.

The most complex expression for aliasing analysis is the function call. To compute possible aliasing because of a function call, the variables propagated by the actual parameters are computed, the pairwise intersection of the sets of variables is performed, and the aliasing information for the appropriate variables is updated (see the definition of $\mathcal{A}$ in figure 4.b). The variables propagated by each expression can be computed in constant time assuming table lookup. The time complexity of pairwise intersection of k ordered sets of size k each is $O(k^3)$. There can be at most k variables whose aliasing information is updated. The aliasing environment update can be done in $O(k)$ time assuming an array representation for environments. Therefore, the complexity of analyzing a function call is bounded by $O(k^3)$. Since there are m non LET-expressions in the entire program, the time complexity of one iteration is $O(mk^3)$.

Combining the bound on the number of iterations and the complexity of each iteration, we derive the complexity of $\mathcal{A}_p$ as $O(mnk^5)$.

8.3 Complexity of $\mathcal{S}_p$

Given a program, $\mathcal{S}_p$ computes an environment in which each function symbol is mapped to an abstract function which returns an ordered pair whose first component is the set of variables selected by the expression evaluation and the second component is the set of variables updated. Each abstract function can be represented as a pair of ordered sets of formal parameters. 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. Therefore each abstract function can take at most k^2 ascending values. The number of iterations needed to reach the top element of $SEnv$ starting with the least element, assuming that at least one function symbol ascends the chain of approximations in each iteration, is $O(nk^2)$. This gives us a bound on the number of iteration for fixpoint calculation.

For computing the complexity of each iteration, we analyze the complexity of analyzing a function call which is the most complex expression for analysis. The set of variables propagated by each argument is at most k. The worst case function abstraction is the lub of k pairs of abstract values where each pair consists of two sets of k elements. The complexity of one lub operation is $O(k)$ which means that the complexity of taking lub of k pairs is $O(k^2)$. The number of non LET-expressions in the program is m and each of these expressions can take

at most $O(k^2)$ time. Therefore the complexity of analyzing all the non LET-expressions in the program is $O(mk^2)$. The sets of aggregates selected and updated in a function body is obtained by determining these sets for each expression in the body and taking their lub. Since a function body can have at most $\lceil m/n \rceil$ expressions, the time for computing the lub of these is $O(\lceil m/n \rceil k)$. Since there are at most n functions, the complexity of one iteration is $O(mk^2 + nk(\lceil m/n \rceil))$ which can be simplified as $O(mk^2)$ because $n \leq m$. The complexity of the function $\mathcal{S}_p$ is $O(mnk^4)$.

8.4 Complexity of $\mathcal{R}_p$

Given a program, $\mathcal{R}_p$ computes the abstract reference environment. 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.

We now estimate the complexity of each iteration which uses the function $\mathcal{R}$. Again the most complex expression is the function call. First, the variables propagated by each actual parameter of the call are computed by a simple table lookup. Global liveness of a parameter can be checked by looking up the values of the variables propagated by an actual parameter. Since an actual parameter can propagate at most k variables and environment lookup takes constant time (using array representation for environments), the global liveness check can be performed in $O(k)$ time. Local liveness of each parameter can be checked by intersecting the variables propagated by a parameter with the set of live variables. The ordered set intersection of two sets of size at most k can be performed in $O(k)$ time. Thus, liveness of the actual parameter of a function can be tested in $O(k)$ time. Computing the liveness of all the parameters takes $O(k^2)$ time. From these time estimates, the complexity of $\mathcal{R}$ on a function call is $O(k^2)$. The complexity of analyzing the whole program in one iteration is $O(mk^2)$. The overall complexity of $\mathcal{R}_p$ is $O(mnk^3)$.

In deriving the complexity of the reference count analysis, we have assumed the availability of the set of live variables at each LET-binding. This information can be computed in a separate pass before doing the reference count analysis. Since there are at most k variables in any function, the live variable set at any program point can at most have k variables. Given a LET-expression containing j LET-bindings, the live variable set at the i^{th} binding is computed as the union of variables used in the expressions to be evaluated (see the definition of $\mathcal{R}$ in figure 6.b). The variables used in an expression are the variables (formal parameters) syntactically appearing in the expression and the variables propagated by the temporary variables appearing in the expression. The set of variables used in any expression can be computed in $O(k^2)$ time. The time for computing the variables used in all the expressions in the program is $O(mk^2)$. The set of live variables at each LET-binding can be computed using the suffix sum algorithm with set union as the associative operation. The complexity of the suffix sum operation is linear in the number of LET-bindings times the complexity of the union operation—*i.e.* $O(mk)$. The effect

Program	No. of upds	No. of destructive upds		
		no assumed ordering	left-to-right	right-to-left
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

Table 1: Performance of Update Analysis Algorithm

of aliasing is taken into account by replacing each live variable with its aliases which can be performed in $O(mk^2)$ time.

We have shown that all our analyses can be performed in polynomial time. The average running times are much better because most functions do not propagate all their arguments, and therefore the cardinalities of sets involved would be smaller than k. Since there is almost no aliasing among formal parameters of functions for most functional programs, therefore the number of iterations needed for aliasing analysis is much smaller than the worst case estimate.

9 Results

We implemented the above algorithm in Standard ML and tested a few example programs which use flat aggregates. Two dimensional arrays are represented as one dimensional arrays. The programs chosen are matrix multiplication, matrix transpose, gaussian elimination, LU-decomposition, quicksort, bubble sort, counting sort[2], array initialization, and two artificial programs c_1 and c_2. The results of the analysis are shown in the table 1. 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 assumed ordering'. The next two columns show the results of analysis with two fixed orders of evaluation.

[2]the range of numbers is known

We observed that in some of the programs, the order of evaluation information is the key to making all the updates destructive. The three programs matmul, transpose and bubblesort use the function *swap* which is defined as

$$swap\ a\ i\ j \quad = \quad upd(upd(a, i, sel(a, j)), j, sel(a, i))$$

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 [Blo89] assumes that the *upd* operator is evaluated from right to left. Similarly, in the gaussian elimination program (see appendix), which takes arrays A and B and computes X such that AX=B, the order of evaluation of expressions in the 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 table. For the programs quicksort, init, and recursive-fft, any evaluation-order is good for destructive updating. The example program c_1 shows our analysis is able to find an ordering which interleaves the evaluation of arguments of two different function calls. The program c_1 is,

$$\begin{aligned} f\ x\ y\ i &= sel(x, y + i) \\ g\ x\ y\ i &= f(upd(y, i, i), sel(x, i), sel(y, 2 * i)) + f(upd(x, i, i), sel(y, i), sel(x, 2 * i)) \end{aligned}$$

In this example, both the updates can be performed destructively only if all the selects are evaluated before the updates. Because we convert every expression into a LET-expression, the intermediate form for the above function is a LET-expression containing eight subexpressions. Then an order of evaluation can be derived.

The last example program c_2 is the example discussed in section 3. The program is

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

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

10 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 [Set82]. 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. The notion of globalizing a variable was first defined by Schmidt [Sch85] who gave the syntactic criteria for converting the store argument of the direct semantics of a language, with while-loops, into a global variable with all the updates performed destructively. The motivation for this work was to implement the denotational definitions efficiently. This work was generalized as the globalization of function parameters by Sestoft[Ses88]. When a function is called, the cost of the function call depends on the number of arguments to the function. A function can be implemented more efficiently if some of its parameters can be converted into global variables. This reduces the size of the activation record for the function. Pushing an actual parameter on to the stack becomes an assignment to a global variable. This work also assumes a fixed order of evaluation of expressions.

The initial work on call-by-value functional languages is Hudak's abstract reference counting technique for a first-order language with flat aggregates [Hud86]. This work assumes a fixed order of evaluation of expressions. Our work can be thought of as its generalization since we derive an order of evaluation. Our abstraction functions are much simpler than the ones used in [Hud86], which mimics the operational semantics of the language by allocating locations, keeping track of the reference counts etc. Gopinath [Gop88] 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 [Hud86]. Again, this work also does not consider reordering expressions because it assumes liveness has been done by Hudak's reference counting technique. The SAL language has other array creation operators like *cat*, the array concatenation operator, which we haven't considered in our language. Bloss [Blo89] extended the work on update analysis to first-order lazy functional languages which are more difficult to analyze because it is not known when an expression is evaluated. She defines a non-standard semantics called path semantics [Blo89] which gives the information about all possible orders of the evaluation of variables in a program. The 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 [Blo89]. This work also assumes a fixed order of evaluation of strict operators. To the best of our knowledge, our update analysis algorithm is the first one with polynomial time complexity, as Hudak's paper does not discuss the complexity of his analysis.

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 propagation of an aggregate becomes more complex. The aliasing analysis described in this paper has to be

generalized to sharing analysis. The other would be 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 of all the functions in the program.

References

[Abr87] S. Abrahamsky, editor. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd., 1987.

[AHU74] A. Aho, J. Hopcroft, and J.D. Ullman. *Design and Analysis of Computer Algorithms.* Addison Wesley Publishing Company, 1974.

[Blo89] A Bloss. *Path Analysis and Optimization of Non-strict Functional Languages.* PhD thesis, Yale University, Dept. of Computer Science, 1989.

[Gop88] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Stanford University, Computer Systems Laboratory, 1988.

[Hud86] Paul Hudak. A semanitc model of reference counting and its abstraction. In *Proceedings of the 1986 Conference on Lisp and Functional Programming*, 1986.

[Sch85] David Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299:310, 1985.

[Ses88] Peter Sestoft. Replacing function parameters with global variables. Master's thesis, DIKU, University of Copenhagen, October 1988.

[Set82] R Sethi. Pebble games for studying storage sharing. *Theoretical Computer Science*, 19(1):69–84, July 1982.

Appendix

** Matrix Multiplication **

```
{
xchange a i j = upd(upd(a,i,sel(a,j)),j,sel(a,i));

xchange_row a i j n =
 if j = n then a
 else
     xchange_row(xchange(a, n*i + j, n*j + i),i,j+1,n)
 endif;

transpose a i n  =
 if (i=n-1) then a
 else
     transpose(xchange_row(a,i,i+1,n),i+1,n)
 endif;

dotproduct a b i j k acc n =
 if (k = n) then acc
 else
     dotproduct(a,b,i,j,k+1,acc + sel(a,n*i+k)*sel(b,n*i+k),n)
 endif;

help_mmult a b c i j n =
 if i=n then c
 else
     if j = n then help_mmult(a,b,c,i+1,0,n)
     else
         help_mmult(a,b,upd(c,n*i+j,dotproduct(a,b,i,j,0,0,n)),i,j+1,n)
     endif
 endif;

mat_mul a b c n = help_mmult(a, transpose(b,0,n),c,0,0,n);
}
```

** Gaussian elimination **

```
{
normalize a i k l n  =
 if (l = n) then a
 else
     normalize(upd(a,n*i+l,k*sel(a,n*i+l)),i,k,l+1,n)
 endif;

elementary_row_op a i j k l n =
 if l = n then a
 else
     elementary_row_op(upd(a,(n*j+l),k*sel(a, (n*i + l))+sel(a,n*j+l)),
            i,j,k,l+1,n)
 endif;

accumulate a x i j n tmp =
 if (j = n) then tmp
 else
     accumulate(a,x,i,j+1,n,(tmp + sel(a,n*i+j)*sel(x,j)))
 endif;

change_barray a b i  k n =
 if (k=n) then b
 else
     change_barray(a,upd(b,k,(sel(b,k) - sel(b,i)*sel(a,n*i+k))),i,k+1,n)
 endif;

eliminate_i_var a i k n =
 if k = n then a
 else
     eliminate_i_var(elementary_row_op(a,i,k,(0 - sel(a,n*k+i)),i,n),
i,k+1,n)
 endif;

back_substitute a b x i n =
 if (i < 0) then x
 else
     back_substitute(a,b,upd(x,i,sel(b,i) - accumulate(a,x,i,i+1,n,0)),i-1,n)
```

```
 endif;

gauss_eliminate a b i n x =
 if (i = n - 1) then
   back_substitute(a,b,upd(x,n-1,sel(b,n-1)/sel(a,n*n-1)),n-2,n)
 else
     gauss_eliminate(eliminate_i_var(
 normalize(a,i,1/sel(a,n*i+i),i,n),i,i+1,n),
            change_barray(a,upd(b,i,(0 - sel(b,i)/sel(a,n*i+i))),i,i+1,n),
    i+1,n,x)
 endif;
}
```

** LU-decomposition **

```
{
lu_decomp a i n =
 if i = n then a
 else
     lu_decomp(compute_shur_complement(divide_column(a,i,i+1,sel(a,n*i+i),n),i+1,n)
                  ,i+1,n)
 endif;

divide_column a i j k n =
 if (j = n) then a
 else
     divide_column(upd(a,j*n + i,sel(a, j*n + i)/k),i,j+1,k,n)
 endif;

compute_shur_complement a i n  = help_compute_shur(a,i,i+1,n);

help_compute_shur a i j n  =
 if j = n then a
 else
     help_compute_shur(update_row(a,i,j,i+1,n), i, j+1,n)
 endif;

update_row a i j k n  =
 if k  = n then a
 else
     update_row( upd(a, j*n + k, sel(a, j*n + k) - sel(a,n*j+i)*
                                                   sel(a,n*i+j)),
                                    i,j, k+1,n)
 endif;
}
```

** Quicksort **

```
{
quicksort vec n = qsort vec 1 n;

qsort v left right =
 if left >= right then v
 else
     scanright(v,left+1,right,sel(v,left),left,right)
 endif;

scanright v l r pivot left right =
 if (l = r) then finish(upd(v,l,pivot),l,left,right)
 else
     if sel(v,r) >= pivot then scanright(v,l,r-1,pivot,left,right)
     else
          scanleft(upd(v,l,sel(v,r)),l+1,r,pivot,left,right)
     endif
 endif;

scanleft v l r pivot left right =
 if (l=r) then finish(upd(v,l,pivot),l,left,right)
 else
     if sel(v,l) <= pivot then scanleft(v,l+1,r,pivot,left,right)
     else
          scanright(upd(v,r,sel(v,l)),l,r-1,pivot,left,right)
     endif
 endif;

finish v mid left right = qsort(qsort(v,left,mid-1),mid+1,right);
}
```