Static Analysis for Safe Destructive Updates in a Functional Language*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. Functional programs are more amenable to rigorous mathematical analysis than imperative programs, but are typically less efficient in terms of execution space and time. The update of aggregate data structures, such as arrays, are a significant source of space/time inefficiencies in functional programming. Imperative programs can execute such updates in place, whereas the semantics of functional languages require aggregate data structures to be copied and updated. In many functional programs, the execution of aggregate updates by copying is redundant and could be safely implemented by means of destructive, in-place updates. We describe a method for analyzing higher-order, eager functional programs for safe destructive updates. This method has been implemented for the PVS specification language for the purpose of animating or testing specifications to check if they accurately reflect their intended function. We also give a careful proof of correctness for the safety of the destructive update optimization.

1 Introduction

Unlike imperative programming languages, pure functional languages are referentially transparent so that two occurrences of the same expression evaluate to the same value in the same environment. The execution semantics of functional languages are therefore nondestructive since variables representing aggregate data structures such as arrays cannot be destructively updated. Pure functional

* Funded by NSF Grants CCR-0082560 and CCR-9712383, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-20334. The author is deeply grateful to the programme committee of the 11th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR 01, for the opportunity to present this work. The LOPSTR 01 programme chair, Professor Alberto Pettorossi, made several excellent suggestions, and Pavol Cerny (visiting SRI from ENS Paris) corrected numerous typographical errors in earlier drafts of this paper.

languages do not admit constructs for performing in-place modifications of aggregate data structures. The aggregate update problem for functional programs is that of statically identifying those array updates in a program that can be executed destructively while preserving the semantics of the program. This problem has been widely studied but none of the previously proposed techniques appear to have actually been implemented in any widely used functional language. We present a simple, efficient, and effective method for the static detection of safe destructive updates in a functional language. The method has been implemented for the functional fragment of the specification language PVS [ORS92].[1] This fragment is essentially a strongly typed, higher-order language with an eager order of evaluation. The method can be easily adapted to other functional languages, including those with a lazy evaluation order. The analysis method is *interprocedural*. Each function definition is analyzed solely in terms of the *results* of the analysis of the previously defined functions and not their actual definitions. We also outline a proof of the correctness for the introduction of destructive updates.

PVS is a widely used framework for specification and verification. By optimizing functions defined in the PVS specification language with safe destructive updates, specifications can be executed for the purposes of animation, validation, code generation, and fast simplification. The technique is presented for a small functional language fragment of PVS.

The concepts are informally introduced using a first-order functional language with booleans, natural numbers, subranges, *flat* (unnested) arrays over subranges, application, conditionals, and array updates. A flat array maps an index type that is a subrange type $[0..n]$ to an members of an element type that is either a boolean, natural number, or subrange type. The range type of the mapping cannot be a function or array type.

The full analysis given in Section 2 is for a higher-order language that includes lambda-abstractions. A function is defined as $f(x_1, \ldots, x_n) = e$ where e contains no free variables other than those in $\{x_1, \ldots, x_n\}$. A few simple examples serve to motivate the ideas. Let `Arr` be an array from the subrange `[0..9]` to the integers. Let A and B be variables of type `Arr`. An array lookup is written as $A(i)$ for $0 \leq i \leq 9$. An array update has the form $A[(i) := a]$ and represents a new array A' such that $A'(i) = a$ and $A'(j) = A(j)$ for $j \neq i$. Pointwise addition on arrays $A + B$ is defined as the array C such that $C(i) = A(i) + B(i)$ for $0 \leq i \leq 9$. Now consider the function definition

$$f_1(A) = A + A[(3) := 4].$$

When executing $f_1(A)$, the update to A cannot be carried out destructively since the original array is an argument to the $+$ operation. The evaluation of

[1] The PVS system and related documentation can be obtained from the URL `http://pvs.csl.sri.com`. The presentation in this paper is for a generic functional language and requires no prior knowledge of PVS. The notation used is also somewhat different from that of PVS.

$A[(3) := 4]$ must return a reference to a new array that is a suitably modified copy of the array A.

The implementation of array updates by copying can be expensive in both space and time. In many computations, copying is unnecessary since the original data structure is no longer needed in the computation that follows the update. Consider the definition

$$f_2(A, i) = A(i) + A[(3) := 4](i).$$

Given an eager, left-to-right evaluation order (as defined in Section 3), the expression $A(i)$ will be evaluated prior to the update $A[(3) := 4]$. Since the original value of A is no longer used in the computation, the array can be updated destructively.[2] The optimization assumes that array A is not referenced in the context where $f_2(A, i)$ is evaluated. For example, in the definition

$$f_3(A) = A[(4) := f_2(A, 3)],$$

it would be unsafe to execute f_2 so that A is updated destructively since there is a reference to the original A in the context when $f_2(A, 3)$ is evaluated.

Next, consider the function definition

$$f_4(A, B) = A + B[(3) := 4].$$

Here, the update to array B can be executed destructively provided A and B are not bound to the same array reference. This happens, for instance, in the definition

$$f_5(C) = f_4(C, C).$$

In such a situation, it is not safe to destructively update the second argument C of f_4 when evaluating the definition of f_4 since the reference to C from the first argument is live, i.e., appears in the context, when the update is evaluated.

The task is that of statically analyzing the definitions of programs involving function definitions such as those of f_1, f_2, f_3, f_4, and f_5, in order to identify those updates that can be executed destructively. Our analysis processes each definition of a function f, and generates the definition for a (possibly) destructive analogue f^D of f that contains destructive updates along with the conditions $LA(f^D)$ under which it is safe to use f^D instead of f. The analysis $LA(f^D)$ is a partial map of the form $\langle x_1 \mapsto X_1, \ldots, x_n \mapsto X_n \rangle$ where $\langle\rangle$ is the empty map. The analysis when applied to a definition $f(x_1, \ldots, x_n) = e$ produces a definition of the form $f^D(x_1, \ldots, x_n) = e^D$, where some occurrences of *nondestructive* updates of the form $e_1[(e_2) := e_3]$ in e have been replaced by *destructive* updates of the form $e_1[(e_2) \leftarrow e_3]$. The analysis of the examples above should therefore

[2] With a lazy order of evaluation, the safety of this optimization depends on the order in which the arguments of + are evaluated.

yield

$$\begin{array}{rll} f_1^D(A) = A + A[(3) := 4] & & LA(f_1^D) = \langle\rangle \\ f_2^D(A,i) = A(i) + A[(3) \leftarrow 4](i) & & LA(f_2^D) = \langle A \mapsto \emptyset\rangle \\ f_3^D(A) = A[(4) \leftarrow f_2(A,3)] & & LA(f_3^D) = \langle A \mapsto \emptyset\rangle \\ f_4^D(A,B) = A + B[(3) \leftarrow 4] & & LA(f_4^D) = \langle B \mapsto \{A\}\rangle \\ f_5^D(C) = f_4(C,C) & & LA(f_5^D) = \langle\rangle \end{array}$$

Observe that when the array referenced by B is destructively updated in f_4^D, the variable A is live, and hence $LA(f_4^D) = \langle B \mapsto \{A\}\rangle$. The information in $LA(f_4^D)$ is used to reject $f_4^D(C,C)$ as unsafe in the definition of f_5^D. In general, this kind of interprocedural analysis can be too coarse. definition. For example, the definition $f_6(A,B,C) = (A+B[(3) := 4]+C[(4) := 3])$ can be safely mapped to either $f_6^D(A,B,C) = (A + B[(3) \leftarrow 4] + C[(4) \leftarrow 3])$, where $LA(f_6^D) = \langle B \mapsto \{A\}, C \mapsto \{A,B\}\rangle$. The update analysis rejects $f_6^D(A,A,B)$ as unsafe, even though this would have been safe had $f_6^D(A,B,C)$ been defined as $(A+B[(3) := 4] + C[(4) \leftarrow 3])$.

We now informally describe the conditions under which $f^D(x_1,\ldots,x_n) = e$ together with the liveness analysis $LA(f^D)$, is a safe, destructive counterpart of the definition $f(x_1,\ldots,x_n) = e$. The liveness analysis table $LA(f^D)$ as a partial map from the set of variables $\{x_1,\ldots,x_n\}$ to its powerset such that $x_j \in LA(f^D)(x_i)$ if x_j is live in the context (as defined below) of a destructive update applied to (the value bound to) x_i. The table $LA(f^D)$ can be used to determine whether it is safe to replace $f(a_1,\ldots,a_n)$ by $f^D(a_1,\ldots,a_n)$ in another function definition.

Given a definition $f^D(x_1,\ldots,x_n) = e$, where e contains an occurrence of a destructive update operation of the form $e_1[(e_2) \leftarrow e_3]$, the task is to identify if this is a safe destructive update. The crucial idea here is that when the destructive update expression $e_1[(e_2) \leftarrow e_3]$ is evaluated, the array reference for the value of e_1 is modified. This array reference is either freshly created within e_1, in which case the destructive update is safe, or it appears in the binding of some *free* variable in e_1. In the latter case, the update is unsafe if such a variable in e_1 is *live* in the *context* when the update $e_1[(e_2) \leftarrow e_3]$ is being evaluated. More strongly, the analysis must ensure that the value of e_1 does not have any array references in common with its context as it appears when the update is evaluated. Such an analysis can be carried out by examining the *mutable* variables that occur in the body of a function definition. A *mutable type* is a type whose values can contain references. For a first-order language, only array types are mutable. A *mutable variable* is a variable of mutable type.

A specific occurrence of a destructive update u of the form $e_1[(e_2) \leftarrow e_3]$ in an expression e can be identified by decomposing e as $U\{u\}$, where U is an *update context* containing a single occurrence of the *hole* $\{\}$, and $U\{u\}$ is the result of filling the hole with the update expression u. In order to determine if u is a safe destructive update, we compute

1. The set $Lv(U)$ of *live* mutable variables in the update context U. When the expression $U\{u\}$ is evaluated, the free variables in it are bound to values through some substitution σ, and the free mutable variables are bound to values containing references. The set $Lv(U)$ calculates those variables x such that $\sigma(x)$ is present in the partially evaluated context U' when the subexpression u is evaluated. This is a subset of the mutable variables in U.
2. The set $Ov(e_1)$ of the output array variables in e_1 contains those array variables x such that the reference $\sigma(x)$ is a possible value of $\sigma(e_1)$.

A destructive update expression $e_1[(e_2) \leftarrow e_3]$ occurring in an update context U, where $e \equiv U\{e_1[(e_2) \leftarrow e_3]\}$ in a definition $f^D(x_1, \ldots, x_n) = e$, is *safe* if $Lv(U) \cap Ov(e_1) = \emptyset$. Informally, this means that when some instance $\sigma(e)$ of e is evaluated, the array references that are possible values of $\sigma(e_1)$ do not occur in the context derived from $\sigma(U)$ when the destructive update $e_1[(e_2) \leftarrow e_3]$. However, we are assuming that whenever $x_i \not\equiv x_j$, then $\sigma(x_i) \neq \sigma(x_j)$. This assumption is violated when *aliasing* occurs, i.e., when $\sigma(x_i) = \sigma(x_j)$ for $x_i \not\equiv x_j$, as is the case in the evaluation of $f_4(C, C)$ in the definition of f_5. To account for the possibility aliasing, a table $LA(f^D)$ is constructed so that $Lv(U) \subseteq LA(f^D)(x)$ for each x in $Ov(e_1)$, i.e., $LA(f^D)(x)$ is the set of variables that must not be aliased to x in any invocation of f^D.

The analysis can use the table $LA(g^D)$ to determine when it is safe to invoke a destructive function application $g^D(a_1, \ldots, a_m)$ within the definition of f^D. An application occurrence $g^D(a_1, \ldots, a_m)$ in an update context U, where $U\{g^D(a_1, \ldots, a_m)\} \equiv e$, is safe iff $Ov(a_i) \cap (Lv(U) \cup Ov(a_j)) = \emptyset$ for each x_i in the domain of $LA(g^D)$ and x_j in $LA(g^D)(x_i)$. Why does this condition ensure the safety of the given occurrence of $g^D(a_1, \ldots, a_m)$? If $\sigma(e)$ is the instance of e that is being evaluated, the references that are destructively updated in evaluating $\sigma(g^D(a_1, \ldots, a_m))$ are from the values of $\sigma(a_i)$ for x_i in the domain of $LA(g^D)$. During the evaluation of the body of the definition of g^D, there can be a destructive update to a reference in the value of $\sigma(a_i)$ for x_i in the domain of $LA(g^D)$. The references in the context of this destructive update are either those from the context $\sigma(Lv(U))$ or from $\sigma(Ov(a_j))$ for $x_j \in LA(g^D)(x_i)$. The mapping $LA(f^D)$ must then be defined to satisfy the constraint $Lv(U) \cup Ov(a_j) \subseteq LA(f)(x)$ for each x in $Ov(a_i)$ such that x_i is in the domain of $LA(g^D)$ and x_j in $LA(g^D)(x_i)$.

Thus in the examples f_1 to f_5, we have

1. f_1: If e^D is $A{+}A[(3) \leftarrow 4]$, then $Ov(A)$ is $\{A\}$, the update context is $(A{+}\{\})$, and $Lv(A + \{\})$ is $\{A\}$. Since $Ov(A)$ has a nonempty intersection with the live variables in the context, the update is not safe.
2. f_2: If e^D is $A(i) + A[(3) \leftarrow 4](i)$, $Ov(A)$ is $\{A\}$, then the update context is $(A(i) + \{\}(i))$, and $Lv(A(i) + \{\}(i))$ is $\emptyset$. Since the updated variable A is not live in the update context, the update is safe. Note that $LA(f_2^D)(A) = \emptyset$.
3. f_3: If e^D is $A[(4) \leftarrow f_2^D(A, 3)]$. Here, the update context for $f_2^D(A, 3)$ is $(A[(4) \leftarrow \{\}])$, where $Lv(A[(4) \leftarrow \{\}])$ is $\{A\}$. Since A is in the domain of $LA(f_2^D)$ and there is a nonempty intersection between $Ov(A)$ and the live

variable set $\{A\}$, the occurrence of $f_2^D(A,3)$ is unsafe. The update $A[(4) \leftarrow f_2(A,3)]$ can be executed destructively, since there are no live references to A in the update context $\{\}$. We then have $LA(f_3^D)(A) = \emptyset$.

4. f_4: If e^D is $A + B[(3) \leftarrow 4]$, then $Ov(B)$ is $\{B\}$, the update context is $(A + \{\})$, and $Lv(A + \{\})$ is $\{A\}$. Since $\{B\} \bigcap \{A\} = \emptyset$, the update is safe, but $LA(f_4^D)(B) = \{A\}$.
5. f_5: If e^D is $f_4^D(C,C)$, then the update context is $\{\}$ with $Lv(\{\})$ equal to $\emptyset$. Since $LA(f_4^D)$ maps B to $\{A\}$ and $Ov(C) = \{C\}$, the analysis detects the aliasing between the binding C for A and C for B. The occurrence of $f_4^D(C,C)$ is therefore unsafe.

The formal explanation for the destructive update analysis and optimization is the topic of the remainder of the paper. A similar analysis and transformation for safe destructive updates was given independently and earlier by Wand and Clinger [WC98] for a first-order, eager, functional language with flat arrays. Their work is in turn based on a polynomial-time, interprocedural analysis given by Shastry, Clinger, and Ariola [SCA93] for determining safe evaluation orders for destructive updates. In this paper, we go beyond the treatment of Wand and Clinger by

1. Employing a notion of update contexts in presenting the analysis.
2. Simplifying the proof of correctness through the use of evaluation contexts.
3. Applying the optimization to a richer language with higher-order operations.
4. Carrying out a complexity analysis of the static analysis procedure.

These extensions are the novel contributions of the paper. We have also implemented our method as part of a code generator for an executable functional fragment of the PVS specification language which generates Common Lisp programs. It was released with PVS 2.3 in the Fall of 1999. Functional programs, such as sorting routines, written in this fragment of PVS execute at speeds that are roughly a factor of five slower than the corresponding programs written in C, and with comparable space usage. The slowness relative to C is primarily due to the overhead of dealing dynamically with multiple array representations.

Clean [vG96] is a lazy functional language where a destructive update optimization has been implemented. The Clean optimization scheme requires programmer annotations to ensure safe destructive updates. Other annotation schemes for expressing destructive updates in functional languages include the use of state monads [Wad97] and various linear type systems [Wad90]. The method presented here does not rely on any programmer annotations.

There is a large body of work on static analysis applied to the destructive array update problem including that of Hudak [Hud87], Bloss and Hudak [BH87], Bloss [Blo94], Gopinath and Hennessy [GH89], and Odersky [Ode91]. Draghicescu and Purushothaman [DP93] introduce the key insight exploited here that in a language with flat arrays, the sharing of references between a term and its context can only occur through shared free variables, but their update analysis for

a lazy language has exponential complexity. Laziness complicates the analysis since there is no fixed order of evaluation on the terms as is the case with eager evaluation. Goyal and Paige [GP98] carry out a copy optimization that works together with reference counting to reduce the need for copying data structures in the set-based language SETL. The *destructive evaluation* of recursive programs studied by Pettorossi [Pet78] and Schwarz [Sch82], employs annotations to construct an intermediate form that explicitly reclaims storage cells during evaluation. This is a slightly different problem from that of destructive array updates, but their annotations are obtained through a similar live variable analysis.

In comparison to previous approaches to update analyses, the method given here is simple, efficient, interprocedural, and has been implemented for an expressive functional language. The implementation yields code that is competitive in performance with efficient imperative languages. The proof of correctness is simple enough that the method can be adapted to other languages with only modest changes to the correctness argument.

2 Update Analysis

We describe a small functional language and an update analysis procedure for this language that generates a destructive counterpart to each function definition. The language is strongly typed. Each variable or function has an associated type. Type rules identify the well-typed expressions. The type of a well-typed expression can be computed from the types of its constituent subexpressions. Types are exploited in the analysis, but the principles apply to untyped languages as well.

The base types consist of `bool`, `integer`, and *index types* of the form $[0 < \kappa]$, where κ is a numeral. The only type constructor is that for function types which are constructed as $[T_1, \ldots, T_n \rightarrow T]$ for types $T, T_1, \ldots, T_n$. The language admits subtyping so that $[0 < i]$ is a subtype of $[0 < j]$ when $i \leq j$, and these are both subtypes of the type `integer`. A function type $[S_1, \ldots, S_n \rightarrow S]$ is a subtype of $[T_1, \ldots, T_n \rightarrow T]$ iff $S_i \equiv T_i$ for $0 < i \leq n$, and S is a subtype of T. We do not explain more about the type system and the typechecking of expressions. Readers are referred to the formal semantics of PVS [OS97] for more details. An array type is a function type of the form $[[0 < i] \rightarrow W]$ for some numeral i and base type W, so that we are, for the present, restricting our attention to flat arrays. The language used here is similar to that employed by Wand and Clinger [WC98], but with the important inclusion of higher-order operations and lambda-abstraction. We allow arrays to be built by lambda-abstraction, whereas Wand and Clinger use a `NEW` operation for constructing arrays.

The metavariable conventions are that W ranges over base types, S and T range over types, x, y, z range over variables, p ranges over primitive function sym-

bols, f and g range over defined function symbols, a, b, c, d, and e range over expressions, L, M, N range over sets of array variables.

The expression forms in the language are

1. Constants: Numerals and the boolean constants `TRUE` and `FALSE`.
2. Variables: x
3. Primitive operations p (assumed to be nondestructive) and defined operations f.
4. Abstraction: $(\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)$, is of type $[T_1, \ldots, T_n \rightarrow T]$, where e is an expression of type T given that each x_i is of type T_i for $0 < i \leq n$. We often omit the types $T_1, \ldots, T_n$ for brevity.
5. Application: $e(e_1, \ldots, e_n)$ is of type T where e is an expression of type $[T_1, \ldots, T_n \rightarrow T]$ and each e_i is of type T_i.
6. Conditional: `IF` e_1 `THEN` e_2 `ELSE` e_3 is of type T, where e_1 is an expression of type `bool`, and e_2, and e_3 are expressions of type T.
7. Update: A nondestructive update expression $e_1[(e_2) := e_3]$ is of type $[[0 < i] \rightarrow W]$, where e_1 is of array type $[[0 < i] \rightarrow W]$, e_2 is an expression of type $[0 < i]$, and e_3 is an expression of type W. A destructive update expression $e_1[(e_2) \leftarrow e_3]$ has the same typing behavior as its nondestructive counterpart.

A program is given by a sequence of function definitions where each function definition has the form $f(x_1 : T_1, \ldots, x_n : T_n) : T = e$. The body e of the definition of f cannot contain any functions other than the primitive operations, the previously defined functions in the sequence, and recursive occurrences of f itself. The body e cannot contain any free variables other than those in $\{x_1, \ldots, x_n\}$.

A variable of array type is bound to an array reference, i.e., a reference to an array location in the store, as explicated in the operational semantics (Section 3). A type is *mutable* if it is an array or a function type. A type is *updateable* if it is an array type or a function type whose range type is updateable. A variable is mutable if its type is mutable, and updateable if its type is updateable. $Mv(a)$ is the set of all mutable free variables of a, $Ov(a)$ is the set of updateable output variables in a, and $Av(a)$ is the set of active mutable variables in the value of a. These will be defined more precisely below so that $Ov(a) \subseteq Av(a) \subseteq Mv(a)$.

Output Analysis. In order to analyze the safety of a destructive update, we need to compute the set of variables whose references could be affected by the update. For an update expression $e_1[(e_2) \leftarrow e_3]$, this is just the set of variables in e_1 that might potentially propagate array references to the value. $Mv(a)$ is defined as the set of mutable free variables of a. The set of output variables of an expression a of array type is computed by $Ov(a)$. Thus $Ov(a)$ could be conservatively approximated by the set of *all* updateable variables in a, but the analysis below is more precise. The auxiliary function $Ovr(a)$ computes a lambda-abstracted set of variables $(\lambda(x_1, \ldots, x_n) : S)$ for a defined function or a lambda-abstraction in the function position of an application. This yields a

more precise estimate of the set of output variables. For example, if the array addition operation + is defined as $(\lambda X, Y : (\lambda(x : [0 < i]) : X(x) + Y(x)))$, then $Ovr(X + Y) = (\lambda(X, Y) : \emptyset)(\{X\}, \{Y\}) = \emptyset$.

Given a sequence of definitions of functions $f_1, \ldots, f_m$, the table OA is a map from the function index i to the output analysis for the definition of f_i, i.e., $Ovr(f_i) = OA(i)$. A map such as OA from an index set I to some range type T is represented as $\langle i_1 \mapsto t_1, \ldots, i_n \mapsto t_n \rangle$. The domain of a map OA is represented as $dom(OA)$. The result of applying a map OA to a domain element i is represented as $OA(i)$. The update of a map OA as $OA\langle i \mapsto t \rangle$ returns t when applied to i, and $OA(j)$ when applied to some j different from i. The empty map is just $\langle\rangle$. For a sequence of definitions of functions $f_1, \ldots, f_m$, the output analysis table OA is defined as OA_m, where OA_0 is the empty map $\langle\rangle$, and $OA_{i+1} = OA^k_{i+1}$ for the least k such that $OA^k_{i+1} = OA^{k+1}_{i+1}$. The map OA^j_i for the definition $f_i(x_1, \ldots, x_n) = e$, is computed iteratively as

$$\begin{aligned} OA^0_i &= OA_{i-1}\langle i \mapsto (\lambda x_1, \ldots, x_n : \emptyset) \rangle \\ OA^{k+1}_i &= OA^k_i \langle i \mapsto Ovr(OA^k_i)(\lambda(x_1, \ldots, x_n) : e) \rangle. \end{aligned}$$

The over-approximation of the output variables, $Ov(a)$, is defined below in terms of the auxiliary function $Ovr(a)$. The defining equations have to be read in order so that the first equation, when applicable, supersedes the others. The case of destructive updates $e_1[(e_2) \leftarrow e_3]$ in the definition of Ovr is counterintuitive. $Ovr(F)(e_1[(e_2) \leftarrow e_3])$ is defined to return $\emptyset$ instead of $Ovr(F)(e_1)$. This is because the destructive update overwrites the array reference corresponding to the value of e_1, and does not propagate the original array to the output.

$$\begin{aligned} Ovr(F)(a) &= \emptyset, \text{ if } a \text{ is not of updateable type} \\ Ovr(F)(x) &= \{x\}, \text{ if } x \text{ is of updateable type} \\ Ovr(F)(f_i) &= F(i) \\ Ovr(F)(a(a_1, \ldots, a_n)) &= Ovr(F)(a)(Ov(F)(a_1), \ldots, Ov(F)(a_n)) \\ Ovr(F)(\lambda(x : [0 < i]) : e) &= \emptyset \\ Ovr(F)(\lambda(x_1, \ldots, x_n) : e) &= (\lambda(x_1, \ldots, x_n) : Ov(F)(e)) \\ Ovr(F)(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Ov(F)(e_2) \cup Ov(F)(e_3) \\ Ovr(F)(a_1[(a_2) := a_3]) &= \emptyset \\ Ovr(F)(a_1[(a_2) \leftarrow a_3]) &= \emptyset \\ Ov(F)(a) &= S - \{x_1, \ldots, x_n\}, \text{ if} \\ & \qquad Ovr(F)(a) = (\lambda(x_1, \ldots, x_n) : S) \\ Ov(F)(a) &= Ovr(F)(a), \text{ otherwise} \end{aligned}$$

When F is fixed to be OA, we just write $Ovr(e)$ for $Ovr(OA)(e)$, and $Ov(e)$ for $Ov(OA)(e)$. Note that $Ovr(e)$ can return either a lambda-abstraction $(\lambda x_1, \ldots, x_n : S)$ or a set of variables S. The definition above uses the application form

$$Ovr(F)(a)(Ov(F)(a_1), \ldots, Ov(F)(a_n))$$

which is defined below.

$$(\lambda(x_1,\ldots,x_n):S)(S_1,\ldots,S_n) = (S-\{x_1,\ldots,x_n\}) \cup \bigcup\{S_i | x_i \in S\}$$
$$S(S_1,\ldots,S_n) = S \cup S_1 \cup \ldots \cup S_n$$

As an example, we extract the output variables returned by the definition

$$f_1(x,y,z,i) = \texttt{IF}(x(i) < y(i), x, f_1(y,z,x,i)).$$

The iterations then proceed as

$$\begin{aligned}
OA_1^0(1) &= (\lambda x,y,z,i:\emptyset)\\
OA_1^1(1) &= (\lambda x,y,z,i:\{x\})\\
OA_1^2(1) &= (\lambda x,y,z,i:\{x,y\})\\
OA_1^3(1) &= (\lambda x,y,z,i:\{x,y,z\})\\
OA_1^4(1) &= (\lambda x,y,z,i:\{x,y,z\})
\end{aligned}$$

The complexity of computing the output variables of an expression e with n updateable variables is at most $n * |e|$ assuming that the set operations can be performed in linear time by representing the sets as bit-vectors. If the size of these bit-vectors fits in a machine word, then the set operations take only constant time and the complexity is just $|e|$. Since the cardinality of the output variables of an expression is bounded by the number n of updateable variables in the expression, the fixed point computation of the table entry OA has at most n iterations. The complexity of computing the table entry for a function definition is therefore at most $n^2 * |e|$.

Active Variables. The set $Av(a)$ of variables returns the active variables in an expression. It is used to keep track of the variables that point to active references in already evaluated expressions. The set $Av(a)$ includes $Ov(a)$ but additionally contains variables that might be trapped in closures. We already saw that $Ov(\lambda(x:[0<i]):X(x)+Y(x)) = \emptyset$ since this returns a new array reference. Also, $Ov(\lambda(x:\texttt{integer}):X(i)+x)$ (with free variables X and i) is $\emptyset$ because the lambda-abstraction is not of updateable type. However, the evaluation of $(\lambda(x:\texttt{integer}):X(i)+x)$ yields a value that traps the reference bound to X. This means that X is live in a context that contains $(\lambda(x:\texttt{integer}):X(i)+x)$. On the other hand, $Av(a)$ is more refined than $Mv(a)$ since $Av(X(i))$ is $\emptyset$, whereas $Mv(X(i))$ is X.

As with the output variables, the active variable analysis for the defined operations $f_1,\ldots,f_m$, are computed and stored in a table VA, where $VA = VA_m$, $VA_0 = []$, and $VA_{i+1} = VA_{i+1}^k$ for the least k such that $VA_{i+1}^{k+1} = VA_{i+1}^k$. The computation of VA_i^j is given by

$$\begin{aligned}
VA_i^0 &= VA_{i-1}\langle i \mapsto (\lambda(x_1,\ldots,x_n):\emptyset)\rangle\\
VA_i^{j+1} &= VA_i^j\langle i \mapsto Avr(VA_i^j)((\lambda(x_1,\ldots,x_n):e))\rangle
\end{aligned}$$

The operation $Av(F)(e)$ of collecting the active variables in an expression e relative to the table F is defined in terms of the auxiliary operation $Avr(F)(e)$ as

$$\begin{aligned}
Avr(F)(a) &= \emptyset, \text{ if } a \text{ is not of mutable type} \\
Avr(F)(x) &= \{x\}, \text{ if } x \text{ is of mutable type} \\
Avr(F)(f_i) &= F(i) \\
Avr(F)(a(a_1, \ldots, a_n)) &= Avr(F)(a)(Av(F)(a_1), \ldots, Av(F)(a_n)) \\
Avr(F)((\lambda(x : [0 < i]) : e)) &= \emptyset \\
Avr(F)((\lambda(x_1, \ldots, x_n) : e)) &= (\lambda(x_1, \ldots, x_n) : Av(F)(e)) \\
Avr(F)(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Av(F)(e_2) \cup Av(F)(e_3) \\
Avr(F)(a_1[(a_2) := a_3]) &= \emptyset \\
Avr(F)(a_1[(a_2) \leftarrow a_3]) &= \emptyset \\
Av(F)(f_i) &= \emptyset \\
Av(F)((\lambda(x_1, \ldots, x_n) : e)) &= Mv((\lambda(x_1, \ldots, x_n) : e)) \\
Av(F)(a) &= Avr(F)(a), \text{ otherwise}
\end{aligned}$$

For example, if f_2 is defined as

$$f_2(x, y, z) = \texttt{IF}(z = 0, (\lambda(u : \texttt{integer}) : x(u + u)), f_2(y, x, z - 1)),$$

then $VA_2 = \lambda x, y, z : \{x, y\}$. Given the table VA, we abbreviate $Av(VA)(e)$ as $Av(e)$, and $Avr(VA)(e)$ as $Avr(e)$.

Lemma 1. *$Ov(e) \subseteq Av(e)$.*

Proof. This is easily established since $Ov(e)$ and $Av(e)$ have similar definitions but the latter collects both updateable and mutable variables, whereas the former collects only the updateable variables. For the case of lambda-abstractions e occurring in non-function positions $Av(e)$ collects all the mutable variables, whereas $Ov(e)$ collects only those variables that might be propagated to the output when the lambda-abstraction is actually applied. ∎

The complexity analysis for the computation of the active variables is similar to that for output variables but with n representing the number of mutable variables in the expression.

Update Contexts. An *update context* U is an expression containing a single occurrence of a hole $\{\}$. An update context U has one of the forms

1. $\{\}$.
2. $\{\}(e_1, \ldots, e_n)$.
3. $e(e_1, \ldots, e_{j-1}, \{\}, e_{j+1}, \ldots, e_n)$.

4. $\texttt{IF}(\{\}, e_2, e_3)$, $\texttt{IF}(e_1, \{\}, e_3)$, or $\texttt{IF}(e_1, e_2, \{\})$.
5. $\{\}[(e_2) := e_3]$, $e_1[(\{\}) := e_3]$, or $e_1[(e_2) := \{\}]$.
6. $\{\}[(e_2) \leftarrow e_3]$, $e_1[(\{\}) \leftarrow e_3]$, or $e_1[(e_2) \leftarrow \{\}]$.
7. $U\{V\}$ for update contexts U and V.

The primary observation about update contexts is that the hole $\{\}$ can occur anywhere except within a lambda-abstraction. Note that the context of evaluation of an update expression within a lambda-abstraction is not easily calculated. For ease of explanation, the definition of update contexts above is conservative in not allowing holes $\{\}$ to occur within lambda-abstractions that occur in function positions of beta-redexes, e.g., let-expressions, even though the context of evaluation for the hole can be exactly determined.

Live Variables. The key operation over update contexts is that of calculating the *live* variables $Lv(U)$. Since the order of evaluation is known, it is easy to determine exactly which subexpressions in the context U will have already been evaluated before the expression in the hole. The live variables in U must contain the active variables $Av(a)$ for those subexpressions a that are evaluated before the hole, and $Mv(b)$ for the subexpressions b that are evaluated subsequent to the hole. More precisely,

$$
\begin{aligned}
Lv(\{\}) &= \emptyset \\
Lv(\{\}(e_1, \ldots, e_n)) &= \bigcup_i Mv(e_i) \\
Lv(e(e_1, \ldots, e_{j-1}, \{\}, \ldots, e_n)) &= Av(e) \cup \bigcup_{i<j} Av(e_i) \cup \bigcup_{i>j} Mv(e_i) \\
Lv(\texttt{IF}(\{\}, e_2, e_3)) &= Mv(e_2) \cup Mv(e_3) \\
Lv(\texttt{IF}(e_1, \{\}, e_3)) &= \emptyset \\
Lv(\texttt{IF}(e_1, e_2, \{\})) &= \emptyset \\
Lv(\{\}[(e_2) := e_3]) &= \emptyset \\
Lv(e_1[(\{\}) := e_3]) &= Mv(e_1) \cup Mv(e_3) \\
Lv(e_1[(e_2) := \{\}]) &= Mv(e_1) \\
Lv(\{\}[(e_2) \leftarrow e_3]) &= \emptyset \\
Lv(e_1[(\{\}) \leftarrow e_3]) &= Mv(e_1) \cup Mv(e_3) \\
Lv(e_1[(e_2) \leftarrow \{\}]) &= Mv(e_1) \\
Lv(U\{V\}) &= Lv(U) \cup Lv(V)
\end{aligned}
$$

The complexity analysis for the live variables computation is at most that of computing the active variables of the expression, i.e., $n * |e|$, since this includes the computation of the active variables of the relevant subexpressions as well.

Liveness Analysis. Given a definition of the form $f^D(x_1, \ldots, x_n) = e^D$, $LA(f^D)$ is a partial map from the updateable variables in $\{x_1, \ldots, x_n\}$ so that $LA(f^D)(x_i)$

is the set of variables that must not aliased to x_i whenever f^D is invoked. The set $LA(f^D)(x_i)$ contains those variables in e^D that are live in some update context U of a destructive update $e_1[(e_2) \leftarrow e_3]$, i.e., $e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\}$, and $x_i \in Ov(e_1)$. The liveness analysis $LA(f^D)$ can be calculated by means of a fixed point computation on the definition so that $LA(f^D) = LA^k(f^D)$ for the least k such that $LA^{k+1}(f^D) = LA^k(f^D)$.

$$
\begin{aligned}
LA^0(f^D)(x_i) = \bot,\ &\text{if} \\
&x_i \text{ is not updateable, or} \\
&\forall\, U, e_1, e_2, e_3 : e \equiv U\{e_1[(e_2) \leftarrow e_3]\} \Rightarrow x_i \notin Ov(e_1),\ \text{and} \\
&\forall\, U, g^D, a_1, \ldots, a_n : \quad e \equiv U\{g^D(a_1, \ldots, a_m)\} \\
&\qquad\qquad \Rightarrow \forall\, j : x_j \in dom(LA(g^D)) : x_i \notin Ov(a_j) \\
LA^0(f^D)(x_i) = L_1 \cup L_2,\ &\text{otherwise, where} \\
L_1 = \{y \mid \exists\, U, e_1, e_2, e_3 : \quad & e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\},\ \text{and} \\
& \wedge\, x_i \in Ov(e_1) \\
& \wedge\, y \in Lv(U)\} \\
L_2 = \{y \mid \exists \quad & U, g^D, a_1, \ldots, a_m : \\
& e^D \equiv U\{g^D(a_1, \ldots, a_m)\} \\
& \wedge\, (\exists\, j, l : \quad x_j \in dom(LA(g^D)) \\
& \qquad \wedge\, x_i \in Ov(a_j) \\
& \qquad \wedge\, x_l \in LA(g^D)(x_j) \\
& \qquad \wedge\, y \in Av(a_l) \cup Lv(U)) \\
LA^{k+1}(f^D)(x_i) = LA^k(f^D)(x_i),\ &\text{if} \\
&\forall\, U, a_1, \ldots, a_n : \quad e^D \equiv U\{f^D(a_1, \ldots, a_n)\} \\
&\qquad\qquad \Rightarrow \forall\, j : x_j \in dom(LA^k(f^D)) \wedge x_i \notin Ov(a_j) \\
LA^{k+1}(f^D)(x_i) = L \cup LA^k(f^D)(x_i)\ &\text{where} \\
L = \{y \mid \exists\, U, a_1, \ldots, a_n : \quad & e^D \equiv U\{f^D(a_1, \ldots, a_n)\} \\
& \wedge\, (\exists\, j, l : \quad x_j \in dom(LA^k(f^D)) \\
& \qquad \wedge\, x_l \in LA^k(f^D)(x_j) \\
& \qquad \wedge\, x_i \in Ov(a_j) \\
& \qquad \wedge\, y \in Av(a_l) \cup Lv(U))
\end{aligned}
$$

An entry in the liveness analysis table is a partial map from the set of variables $\{x_1, \ldots, x_n\}$ to subsets of this set. The number of iterations in the fixed point computation of a liveness analysis entry is at worst n^2. Each iteration is itself of complexity at worst $n * |e^D|$, yielding a complexity of $O(n^3 * |e^D|)$ for a definition of the form $f^D(x_1, \ldots, x_n) = e^D$. In recent work, Pavol Cerny has simplified the definitions of these analyses so that the program e is examined only once to compute an abstract flow analysis table. In practice, only a small fraction of the variables in a function definition are mutable, so this complexity is unlikely to be a significant bottleneck.

Safe Updates. Let $\gamma(e)$ represent the result of repeatedly replacing destructive updates $e_1[(e_2) \leftarrow e_3]$ in e by corresponding nondestructive updates $e_1[(e_2) := e_3]$, and destructive applications $g^D(a_1, \ldots, a_n)$ by $g(a_1, \ldots, a_2)$. It is easy to see that if $e = \gamma(e')$, then $Ov(e) = Ov(e')$ and $Av(e) = Av(e')$.

To obtain the destructive definition $f^D(x_1, \ldots, x_n) = e^D$ and the liveness table $LA(f^D)$, from the definition $f(x_1, \ldots, x_n) = e$, we construct e^D so that $\gamma(e^D) \equiv e$ and e^D is safe. An expression e^D is *safe* if

1. Every occurrence of $e_1[(e_2) \leftarrow e_3]$ in e^D within an update context U (i.e., $e^D \equiv U\{e_1[(e_2) \leftarrow e_3]\}$), satisfies $Ov(e_1) \cap Lv(U) = \emptyset$ and $Lv(U) \subseteq LA(f^D)(x)$ for each variable x in $Ov(e_1)$.
2. Every occurrence of a destructive function application $g^D(a_1, \ldots, a_n)$ in e within an update context U (i.e., $e^D \equiv U\{g^D(a_1, \ldots, a_n)\}$) satisfies $Ov(a_i) \cap (Lv(U) \cup Av(a_j)) = \emptyset$ for each x_i in the domain of $LA(g^D)$ and $y_i \in LA(g^D)(x_i)$. Furthermore, $Lv(U) \cup Av(a_j) \subseteq LA(f^D)(x)$ for each variable x in $Ov(a_i)$ for x_i in the domain of $LA(g^D)$ and $x_j \in LA(g^D)(x_i)$.

There are several different strategies for identifying safe destructive updates in order to obtain the definition $f^D(x_1, \ldots, x_n) = e^D$ from $f(x_1, \ldots, x_n) = e$. We present one such strategy that is quite conservative. The choice of different strategies does not affect the underlying theory and correctness argument. We first compute e^+ from e by converting all occurrences of safe updates in e to destructive form, all recursive function calls $f(a_1, \ldots, a_n)$ to $f^+(a_1, \ldots, a_n)$, and all safe occurrences of other function calls $g(a_1, \ldots, a_m)$ to $g^D(a_1, \ldots, a_m)$. This yields a definition $f^+(x_1, \ldots, x_n) = e^+$ that might be overly aggressive in its recursive calls. Since the liveness analysis for updates and function calls is unaffected by the transition from e to e^+, the only unsafe subterms in the definition e^+ are the recursive calls of the form $f^+(a_1, \ldots, a_n)$. The next step is to recognize and eliminate the unsafe recursive calls from the definition of f^+. Using $LA(f^+)$, we construct e^D by replacing the unsafe recursive calls $f^+(a_1, \ldots, a_n)$ by $f(a_1, \ldots, a_n)$, and the safe recursive calls by $f^D(a_1, \ldots, a_n)$. This yields the definition $f^D(x_1, \ldots, x_n) = e^D$.

Theorem 1 (Safe Definition). *The destructive definition $f^D(x_1, \ldots, x_n) = e^D$ obtained from the nondestructive definition $f(x_1, \ldots, x_n) = e$, is safe.*

Proof. In the definition $f^+(x_1, \ldots, x_n) = e^+$, the destructive updates and the function calls other than f^+ are safe because they depend only on the output variable analysis $Ov(d)$, the active variable analysis $Av(d)$ for subterms d of e, and the liveness analysis $LA(g)$ for functions g other than f or f^+. These update terms therefore continue to remain safe in the definition of f^+.

For each x_i, $LA(f^D)(x_i)$ is $\perp$ or $LA(f^D)(x_i) \subseteq LA(f^+)(x_i)$. Since the definition e^D is constructed from e^+ by replacing the unsafe destructive recursive calls by safe ones with respect to $LA(f^+)$, the definition $f^D(x_1, \ldots, x_n) = e^D$ is safe with respect to $LA(f^D)$. ∎

3 Operational Semantics

We present operational semantics for the languages with destructive updates. We then exhibit a bisimulation between evaluation steps on a nondestructive expression e and its safe destructive counterpart e^D. The concepts used in defining the operational semantics are quite standard, but we give the details for the language used here.

The expression domain is first expanded to include

1. Explicit arrays: $\#(e_0, \ldots, e_{n-1})$ is an expression representing an n-element array.
2. References: $\mathit{ref}(i)$ represents a reference to reference number i in the store. Stores appear in the operational semantics.

A *value* is either a boolean constant, integer numeral, a closed lambda-abstraction $(\lambda x_1, \ldots, x_n : e)$ or a reference $\mathit{ref}(i)$. The metavariable v ranges over values.

An *evaluation context* [Fel90] E is an expression with an occurrence of a hole $[]$ and is of one of the forms

1. $[]$
2. $[](e_1, \ldots, e_n)$
3. $v(v_1, \ldots, v_{j-1}, [], e_{j+1}, \ldots, e_n)$
4. $\mathtt{IF}([], e_2, e_3)$, $\mathtt{IF}(\mathtt{TRUE}, [], e_3)$, or $\mathtt{IF}(\mathtt{FALSE}, e_2, [])$.
5. $e_1[([]) := e_3]$, $e_1[(v_2) := []]$, or $[][(v_2) := v_3]$.
6. $e_1[([]) \leftarrow e_3]$, $e_1[(v_2) \leftarrow []]$, or $[][(v_2) \leftarrow v_3]$.
7. $E_1[E_2]$, if E_1 and E_2 are evaluation contexts.

A *redex* is an expression of one of the following forms

1. $p(v_1, \ldots, v_n)$.
2. $f(v_1, \ldots, v_n)$.
3. $(\lambda(x : [0 < n]) : e)$.
4. $(\lambda(x_1, \ldots, x_n) : e)(v_1, \ldots, v_n)$.
5. $\#(v_0, \ldots, v_{n-1})$.
6. $\mathtt{IF\ TRUE\ THEN}\ e_1\ \mathtt{ELSE}\ e_2$.
7. $\mathtt{IF\ FALSE\ THEN}\ e_1\ \mathtt{ELSE}\ e_2$.
8. $\mathit{ref}(i)[(v_2) := v_3]$.
9. $\mathit{ref}(i)[(v_2) \leftarrow v_3]$.

A *store* is a mapping from a reference number to an array value. A store s can be seen as a list of array values $[s[0], s[1], \ldots,]$ so that $s[i]$ returns the $(i{+}1)$'th element of the list. Let $s[i]\langle i \mapsto v_i \rangle$ represent the array value $\#(w_0, \ldots, v_i, \ldots, w_{n-1})$, where $s[i]$ is of the form $\#(w_0, \ldots, w_i, \ldots, w_{n-1})$. List concatenation is represented as $r \circ s$.

A *reduction* transforms a pair consisting of a redex and a store. The reductions corresponding to the redexes above are

1. $\langle p(v_1, \ldots, v_n), s\rangle \rightarrow \langle v, s\rangle$, if the primitive operation p when applied to arguments $v_1, \ldots, v_n$ yields value v.
2. $\langle f(v_1, \ldots, v_n), s\rangle \rightarrow \langle [v_1/x_1, \ldots, v_n/x_n](e), s\rangle$, if f is defined by $f(x_1, \ldots, x_n) = e$.
3. $\langle (\lambda(x : [0 < n]) : e), s\rangle \rightarrow \langle \#(e_0, \ldots, e_{n-1}), s\rangle$, where $e_i \equiv (\lambda(x : [0 < n]) : e)(i)$, for $0 \leq i < n$.
4. $\langle (\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)(v_1, \ldots, v_n), s\rangle \rightarrow \langle [v_1/x_1, \ldots, v_n/x_n](e), s\rangle$.
5. $\langle \#(v_0, \ldots, v_{n-1}), s\rangle \rightarrow \langle ref(m), s'\rangle$, where $s \equiv [s[0], \ldots, s[m-1]]$ and $s' \equiv s \circ [\#(v_0, \ldots, v_{n-1})]$.
6. $\langle \texttt{IF TRUE THEN } e_1 \texttt{ ELSE } e_2, s\rangle \rightarrow \langle e_1, s\rangle$.
7. $\langle \texttt{IF FALSE THEN } e_1 \texttt{ ELSE } e_2, s\rangle \rightarrow \langle e_2, s\rangle$.
8. $\langle ref(i)[(v_2) := v_3], s\rangle \rightarrow \langle ref(m), s'\rangle$, where

$$\begin{aligned} s &\equiv [s[0], \ldots, s[i], \ldots, s[m-1]] \\ s' &\equiv [s[0], \ldots, s[i], \ldots, s[m]] \\ s[m] &= s[i]\langle v_2 \mapsto v_3\rangle. \end{aligned}$$

9. $\langle ref(i)[(v_2) \leftarrow v_3], s\rangle \rightarrow \langle ref(i), s'\rangle$, where $s_1 \equiv [s[0], \ldots, s[i], \ldots, s[m-1]]$ and $s_2 \equiv [s[0], \ldots, s[i]\langle v_2 \mapsto v_3\rangle, \ldots, s[m-1]]$).

An evaluation *step* operates on a pair $\langle e, s\rangle$ consisting of a closed expression and a store, and is represented as $\langle e, s\rangle \longrightarrow \langle e', s'\rangle$. If e can be decomposed as a $E[a]$ for an evaluation context E and a redex a, then a step $\langle E[a], s\rangle \longrightarrow \langle E[a'], s'\rangle$ holds if $\langle a, s\rangle \rightarrow \langle a', s'\rangle$. The reflexive-transitive closure of $\longrightarrow$ is represented as $\langle e, s\rangle \xrightarrow{*} \langle e', s'\rangle$. If $\langle e, s\rangle \xrightarrow{*} \langle v, s'\rangle$, then the result of the computation is $s'(v)$, i.e., the result of replacing each reference $ref(i)$ in v by the array $s'[i]$. The computation of a closed term e is initiated on an empty store as $\langle e, []\rangle$. The value $eval(e)$ is defined to be $s(v)$, where $\langle e, []\rangle \xrightarrow{*} \langle v, s\rangle$.

4 Correctness

The correctness proof demonstrates the existence of a bisimulation between evaluations of the unoptimized nondestructive program and the optimized program. The key ideas in the proof are:

1. The safe update analysis is lifted from variables to references since the expressions being evaluated do not contain free variables.
2. The safety of an expression is preserved by evaluation.
3. Given a nondestructive configuration $\langle e, s\rangle$ and its destructive counterpart $\langle d, r\rangle$, the relation $s(e) = \gamma(r(d))$ between the two configurations is preserved by evaluation. Recall that the operation $\gamma(a)$ transforms all destructive updates $a_1[(a_2) \leftarrow a_3]$ in a to corresponding nondestructive updates $a_1[(a_2) := a_3]$, and all destructive applications $g^D(a_1, \ldots, a_n)$ to the corresponding nondestructive applications $g(a_1, \ldots, a_n)$.

4. When e and d are values, then the bisimulation ensures that the destructive evaluation returns the same result as the nondestructive evaluation.

The intuitive idea is that the destructive optimizations always occur safely within update contexts during evaluation. When an update context coincides with an evaluation context U, the references accessible in the context are a subset of $Lv(U)$. The safety condition on the occurrences of destructive operations within an update context then ensures that a reference that is updated destructively does not occur in the context. The observation that all destructive operations occur safely within update contexts can be used to construct a bisimulation between a nondestructive configuration $\langle e, s\rangle$ and a destructive configuration $\langle d, r\rangle$ that entails the invariant $s(e) = \gamma(r(d))$. This invariant easily yields the main theorem

$$eval(e) = eval(e^D)$$

for a closed, reference-free expression e.

The application of a store s of the form $[s[0], \ldots, s[n-1]]$ to an expression e, written as $s(e)$, replaces each occurrence of $ref(i)$ in e by $s[i]$, for $0 \leq i < n$.

The definition of safety for an expression e has been given in Section 2 (page 14). A destructive expression is either a destructive update of the form $e_1[(e_2) \leftarrow e_3]$ or a destructive function invocation of the form $g^D(a_1, \ldots, a_n)$. An expression is safe if all destructive expressions occur safely within update contexts. In the update analysis in Section 2, the expressions being analyzed contained variables but no references, but the expressions being evaluated contain references and not variables. The definitions of Ovr, Mv, and Av have to be extended to include references so that

$$Ovr(ref(i)) = Mv(ref(i)) = Av(ref(i)) = \{ref(i)\}.$$

With this change, for a closed term e, the sets returned by $Ov(e)$, $Mv(e)$, and $Av(e)$, consist entirely of references. Recall that $Ov(e) \subseteq Av(e) \subseteq Mv(e)$.

A closed expression d is *normal* if every occurrence of a destructive expression u in d occurs within an update context, i.e., there is some update context U such that $d \equiv U\{u\}$.

Lemma 2. *Normality is preserved during evaluation: if d is a normal, closed term and $\langle d, r\rangle \Longrightarrow \langle d', r'\rangle$, then so is d'.*

Proof. Essentially, U is an update context if the hole $\{\}$ in U does not occur within a lambda-abstraction. Any evaluation context E can be turned into an update context as $E[\{\}]$, but not necessarily vice-versa. It is easy to see that none of the reductions causes a destructive term to appear within a lambda-abstraction. ∎

Let $dom(r)$ for a store r be the set $\{ref(i) | i < |r|\}$. A configuration $\langle d, r\rangle$ is called *well-formed* if d is a normal, closed term and each reference $ref(i)$ occurring in e is in the domain of r, $dom(r)$.

Lemma 3. *The well-formedness of configurations is preserved during evaluation.*

Proof. By Lemma 2, if $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$ and d is a normal, closed term, then so is d'. The only redexes that introduce new references are $\#(v_0, \ldots, v_{n-1})$ and $ref(i)[(v_2) := v_3]$. In either case, the reduction step ensures that the store is updated to contain an entry for the newly introduced reference. ∎

Since we are dealing with flat arrays, the expression $r(d)$ contains no references when $\langle d, r\rangle$ is a well-formed configuration.

Let $\rho(d)$ be the operation of collecting all the references occurring in the expression d.

Lemma 4. *For every reduction $\langle a, r\rangle \rightarrow \langle a', r'\rangle$, $\rho(a') \cap dom(r) \subseteq \rho(a)$. Hence, for each evaluation step $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, $\rho(d') \cap dom(r) \subseteq \rho(d)$.*

Proof. Any references $ref(i)$ in r that are unreachable in a, i.e., $ref(i) \notin \rho(a)$, are also unreachable in a'. This is because for each of the reductions $\langle a, r\rangle \rightarrow \langle a', r'\rangle$, the only references in a' that are not in a are those that are also not in r. ∎

Lemma 5. *If a is a normal, safe, closed expression, $\langle E[a], s\rangle$ is a well-formed configuration, $\langle E[a], s\rangle \longrightarrow \langle E[a'], s'\rangle$, and $Ov(a') - Ov(a)$ is nonempty, then $Ov(a') \cap Lv(E[\{\}])$ and $Av(a') \cap Lv(E[\{\}])$ are both empty.*

Proof. For a redex a and its residual a', $(Ov(a') - Ov(a))$ (alternately, $(Av(a') - Av(a))$) is nonempty only if a is either an array of the form $\#(v_0, \ldots, v_{n-1})$, an update expression $ref(i)[(v_2) := v_3]$, or a destructive update $ref(i)[(a_2) \leftarrow a_3]$. In the first two cases, the reference in the singleton set $Ov(a')$ is fresh and does not occur in $E[\{\}]$. In the third case, since a' is a destructive subexpression of a safe expression $ref(i) \notin Lv(E[\{\}])$. Since $Ov(a') = Av(a') = \{ref(i)\}$, $Ov(a') \cap Lv(E[\{\}])$ and $Av(a') \cap Lv(E[\{\}])$ are both empty. ∎

Lemma 6. *If E is an evaluation context, then $Lv(E[\{\}]) = Mv(E)$.*

Proof. By induction on the structure of an evaluation context. If $E \equiv []$, then $Lv(E[\{\}]) = Mv(E[\{\}]) = \emptyset$. If $E \equiv [](e_1, \ldots, e_n)$, then $Lv(E[\{\}]) = Mv(E)$. When $E \equiv v(v_1, \ldots, v_{j-1}, [], e_{j+1}, \ldots, e_n)$, we use the observation that for any value v, $Mv(v) = Av(v)$, to conclude that $Lv(E[\{\}]) = Mv(E)$. If $E \equiv \texttt{IF}([], e_2, e_3)$, then $Lv(E[\{\}]) = Mv(e_2) \cup Mv(e_3) = Mv(E)$. The remaining cases for conditional expressions and update expressions can be similarly examined. The final case when $E \equiv E_1[E_2]$, we can assume that E_1 is one of the above cases but not the empty context. We know that $Lv(E_1[E_2[\{\}]) = Lv(E_1[\{\}]) \cup Lv(E_2[\{\}])$, which by the induction hypothesis is just $Mv(E)$. ∎

Theorem 2. *If $\langle d, r\rangle$ is a well-formed configuration, $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, and d is safe, then d' is safe.*

Proof. We have to show that every occurrence of a destructive expression u' in d' is safe. Let d be of the form $E[a]$ for some evaluation context E and redex a. Then, d' is of the form $E[a']$ where $\langle a, r\rangle \rightarrow \langle a', r'\rangle$. Given any occurrence of an update expression u' in an update context U' in d', the residual a' either occurs within u', u' occurs within a', or a' occurs within U. This is because none of the redexes can partially overlap a destructive expression.

If a' occurs properly in u', then $d' \equiv U'\{u'\}$, and the following cases arise:

1. u' is of the form $d'_1[d'_2 \leftarrow d'_3]$: Then, if a' occurs in either d'_2 or d'_3, we have that $d'_1 \equiv d_1$, where $u \equiv d_1[(d_2) \leftarrow d_3]$ and $d \equiv U'\{u\}$. Therefore, $Ov(d'_1) = Ov(d_1)$ and u' occurs safely within the update context U' since u occurs safely within U'.
   If a' occurs in d'_1, then $(Ov(d'_1) - Ov(d_1)) \subseteq (Ov(a') - Ov(a))$. Since every evaluation context is an update context, we have an update context V' such that $d'_1 \equiv V'\{a'\}$, $d_1 \equiv V'\{a\}$, and $d \equiv U'\{V'\{a\}\}$. By Lemma 5, if $(Ov(a') - Ov(a))$ is nonempty, then $Lv(U'\{V'\}) \cap (Ov(a') - Ov(a))$ is empty. Since $Lv(U'\{V'\}) = Lv(U') \cup Lv(V')$, it follows that u' is safe in the update context U'.
2. u' is of the form $g^D(b'_1, \ldots, b'_n)$, where a' occurs in b_i for some i, $1 \leq i \leq n$. Then, $u \equiv g^D(b_1, \ldots, b_n)$, where $b_j \equiv b'_j$ for j, $1 \leq j \leq n$ and $i \neq j$. Since, $\langle d, r\rangle$ is a well-formed configuration, $Lv(U') \subseteq dom(r)$. By Lemma 5, we have that $(Ov(b'_i) - Ov(b)) \cap Lv(U')$ is empty, as is $(Ov(b'_i) - Ov(b_i)) \cap Lv(g^D(\ldots, b_{i-1}, \{\}, b_{i+1}, \ldots\}))$, and similarly for $Av(b'_i) - Av(b_i)$. In order to ensure that $g^D(b'_1, \ldots, b'_n)$ occurs safely within U', we have to check that for any x_j in $dom(LA(g^D))$, $Ov(b'_j) \cap Lv(U')$ is empty, and for $k \neq j$ and x_k in $LA(g^D)x_j$, $Av(b'_k) \cap Ov(x_j)$ is empty. Since $U\{g^D(b_1, \ldots, b_n)\}$ is safe, if $j \neq i$, then clearly $Ov(b'_j) = Ov(b_j)$ and $Ov(b'_j) \cap Lv(U')$ is empty. If additionally, $k \neq i$, then $Av(b'_k) = Av(b_k)$ and $Av(b'_k) \cap Av(b_j)$ is also empty. If $j = i$, then we know that $Ov(b'_j) \cap Lv(U')$ is empty. If $k = i$, then $Av(b'_j) \cap Lv(g^D(\ldots, b'_{i-1}, \{\}, b'_{i+1}, \ldots))$ is empty, and since $Ov(b'_j) \subseteq Lv(g^D(\ldots, b'_{i-1}, \{\}, b'_{i+1}, \ldots))$, we have that $Av(b'_k) \cap Ov(b'_j)$ is empty.

If u' occurs (properly or not) within a', then by the syntax of a redex a, one of the following two cases is possible:

1. Redex a is a conditional expression and a' must be either the `THEN` or `ELSE` part of a. Since the argument is symmetrical, we consider the case when u' occurs in the `THEN` part. Then U is of the form $U_1\{\texttt{IF}(b_1, \{U_2\{\}\}, b_3)\}$ and U' is then of the form $U_1\{U_2\{\}\}$. Since $Lv(U_1) \cup Lv(U_2) \subseteq Lv(U)$. Therefore, u' occurs safely in U'.

2. Redex a is of the form $g^D(v_1, \ldots, v_n)$ and is reduced to a' of the form $\sigma(b^D)$, where g^D is defined as $g^D(x_1 \ldots, x_n) = b^D$ and substitution σ is of the form $\langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \rangle$. We have to ensure that any update expression u' occurring in b^D is safe within the update context U' where $d' \equiv U'\{u'\}$. Note that we have already checked the safety of b^D during the generation of the definition of g^D.
   If u' is of the form $\sigma(b_1[(b_2) \leftarrow b_3])$, we have $b^D \equiv V\{b_1[(b_2) \leftarrow b_3]\}$ where $Ov(b_1) \cap Lv(V) = \emptyset$. Since b^D does not contain any references, the result of applying substitution σ to b^D is just $\sigma(V)\{\sigma(e_1)[\sigma(e_2) \leftarrow \sigma(e_3)]\}$. It is easy to check that $Lv(\sigma(V)) = \rho(Lv(V))$. Note that for a value v, $Ov(v) \subseteq Av(v) \subseteq Mv(v) = \rho(v)$. From the update analysis of g, we know that $Ov(b_1) \cap Lv(V) = \emptyset$ and $Lv(V) \subseteq LA(g^D)(x)$ for $x \in Ov(b_1)$. Note that $Ov(\sigma(b_1)) \subseteq \rho(\sigma(Ov(b_1)))$. Since the occurrence of $g^D(v_1, \ldots, v_n)$ is safe with respect to the update context U, and $U' = U\{V\}$,
   (a) $Ov(\sigma(b_1)) \cap Lv(U) = \emptyset$
   (b) $Ov(\sigma(b_1)) \cap \sigma(Lv(V)) = \emptyset$
   Therefore, the destructive update $\sigma(b_1)[(\sigma(b_2)) \leftarrow \sigma(b_3)]$ is safe with respect to the update context U'.
   A similar argument can be used to show that if u' is of the form $f^D(b_1, \ldots, b_n)$ in b^D, it occurs safely within the update context U', where $d' \equiv U'\{u'\}$.

The final case is when u' and a' do not overlap, then a' occurs in U'. Except for the case where a is a destructive update, every other instance of a reduction of a from a redex a in U yielding U', we can check that $(Lv(U') \cap dom(r)) \subseteq Lv(U)$. Clearly, then u' occurs safely in U' since it occurs safely in U. When redex a is a destructive update of the form $ref(i)[(b_2) \leftarrow b_3]$, then by Lemma 6, $ref(i) \notin Lv(E[\{\}])$. If u' occurs in E, then $Mv(u') \subseteq Lv(E[\{\}])$, and hence u' again occurs safely in U'. ∎

The significance of the safety invariant established in Theorem 2 should be obvious. It shows that whenever a destructive update redex $ref(i)[(v_2) \leftarrow v_3]$ is evaluated, it occurs within a context that is both an evaluation context $E[]$ and an update context $E[\{\}]$. Since by Lemma 6, $Lv(E[\{\}]) = Mv(E)$ and by Theorem 2, $ref(i) \notin Lv(E[\{\}])$, the destructive update can be executed safely without any unintended side-effects.

Given that all configurations are well-formed and safe, it is easy to establish the bisimulation between destructive and nondestructive execution. The bisimulation R between a nondestructive configuration $\langle e, s \rangle$ and a destructive configuration $\langle d, r \rangle$ is given by $\exists \pi : e = \gamma(\pi(d)) \wedge (\forall\ j \in dom(r) : s[\pi(j)] = r[j])$, where π is a permutation from $dom(r)$ to $dom(s)$ and $\pi(d)$ is the result of replacing each occurrence of $ref(i)$ in d by $ref(\pi(i))$.

Theorem 3. *If $\langle e, s \rangle$ is a well-formed configuration where e is a nondestructive expression, and $\langle d, r \rangle$ is a well-formed configuration with a safe destructive expression d, then the relation R given by $\lambda\ \langle e, s \rangle, \langle d, r \rangle : \exists \pi : e = \gamma(\pi(d)) \wedge (\forall\ j \in dom(r) : s[\pi(j)] = r[j])$ is a bisimulation with respect to evaluation.*

Proof. By Lemma 2 we already have that for a safe, well-formed configuration $\langle d, r\rangle$, whenever $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$ then $\langle d', r'\rangle$ is a safe, well-formed configuration. If $R(\langle e, s\rangle, \langle d, r\rangle)$ holds, $\langle e, s\rangle \longrightarrow \langle e, s'\rangle$, and $\langle d, r\rangle \longrightarrow \langle d', r'\rangle$, then we can show that $R(\langle e', s'\rangle, \langle d', s'\rangle)$ holds. First note that if there is a π such that $e = \gamma(\pi(d))$, then if $e = E[a]$ and $d = D[b]$, then $E = \gamma(\pi(D))$ and $a = \gamma(\pi(b))$. We also have that $\langle a, s\rangle \rightarrow \langle a', s'\rangle$ where $e' \equiv E[a']$, and $\langle b, r\rangle \rightarrow \langle b', r'\rangle$, where $d' \equiv D[b']$.

If a is a redex of the form $ref(i)[(v_2) := v_3]$ and b is of the form $ref(j)[(v_2) \leftarrow v_3]$, then we know that $\pi(j) = i$. Since $D[b]$ is safe, $ref(j) \notin Mv(D[\{\}])$, and b' is $ref(j)$, while $r'[j]$ is $r[j]\langle v_2 \mapsto v_3\rangle$ and $r'[k] = r[k]$ for $k \neq j$. On the nondestructive side, a' is $ref(m)$, where s' is $s \circ [s(i)\langle v_2 \mapsto v_3\rangle]$. If we let π' be $\pi\langle j \mapsto m\rangle$, then since $s[i] = r[j]$, we also have $s'[m] = r'[j]$. Since $ref(j) \notin Mv(D)$ and $ref(m) \notin Mv(E)$, $\gamma(\pi'(D)) = \gamma(\pi(D)) = E$.

In every other case for the redexes a and b, the destructive and nondestructive evaluation steps are essentially identical, and $\gamma(\pi(b')) = a'$, and hence $\gamma(r'(b')) = s'(a')$. ∎

The main correctness theorem easily follows from the bisimulation proof.

Theorem 4. *If e and d are closed, reference-free terms such that $\gamma(d) \equiv e$, then*

$$eval(d) \equiv eval(e).$$

5 Observations

As already mentioned, our analysis is essentially similar to one given independently and earlier by Wand and Clinger [WC98], but our presentation is based on update contexts, and the correctness proof here is more direct, due to the use of evaluation contexts. Wand and Clinger use an operational semantics based on an abstract machine with a configuration consisting of a label, an environment, a return context, and a continuation.

The worst-case complexity of analyzing a definition $f(x_1, \ldots, x_n) = e$ as described here is $n^3 * |e|$. Our definitions for the various analyses used here, are quite naive, and their efficiency can be easily improved. The procedure requires n^2 iterations in the fixed point computation since $LA(f)$ is an n-element array consist of at most $n - 1$ variables. The complexity of each iteration is at worst $n * |e|$. In practice, this complexity is unlikely to be a factor since only a few variables are mutable. If n is smaller than the word size, then the set operations can be executed in constant time yielding a complexity $n^2 * |e|$.

We have used a simple core language for presenting the ideas. The method can be adapted to richer languages, but this has to be done carefully. For example, nested array structures introduce the possibility of structure sharing within an array where, for example, both $A(1)$ and $A(2)$ reference the same array. Such internal structure sharing defeats our analysis. For example, the update of index

3 of the array at index 2 of A might have the unintended side-effect of updating a shared reference at index 1 of array A. The analysis has to be extended to rule out the nested update of nested array structures. Non-nested updates of nested arrays, such as $A(2)[(3) := 4]$, are already handled correctly by the analysis since we check that A is not live in the context, and the result returned is the updated inner array $A(2)$ and not the nested array A. Other nested structures such as records and tuples also admit similar structure sharing, but type information could be used to detect the absence of sharing.

Allowing array elements to be functional is only mildly problematic. Here, it is possible for references to the original array to be trapped in a function value (closure) as in $A[(2) := (\lambda(x) : x + A(2)(2))]$. It is easy to modify the notion of an update context and the accompanying definitions to handle functional values in arrays.

The analysis method can be adapted to lazy functional languages. Here, an additional analysis is needed to determine for a function $f(x_1, \ldots, x_n) = e$ if an argument x_j might be evaluated after an argument x_i in the body e of f.

PVS functions are translated to destructive Common Lisp operations that are then compiled and executed. We omit performance information due to lack of space. A draft report with performance results for PVS can be obtained from the URL `www.csl.sri.com/shankar/PVSeval.ps.gz`.

Since functional programs require nondestructive arrays for correct operation, we have carried out some experiments with good data structures for this case. The results here are encouraging but outside the scope of this paper.

6 Conclusions.

Experience has shown that the semantic simplicity of functional programming can be used to derive efficient and easily optimizable programs. Optimizations arising from the static analysis for destructive updates presented in this paper enables functional programs to be executed with time and space efficiencies that are comparable to low-level imperative code. The update analysis has been implemented within a code generator for the functional fragment of the specification language PVS. An efficient execution capability for this fragment is useful for validating specifications and in fast simplification of executable terms arising in a proof.

A common criticism of specification languages is that there is a duplication of effort since the design is described in a specification language as well as an implementation language. As a result, the verified design is different from the implemented one, and these two designs evolve separately. Through the generation of efficient code from logic, it is possible to unify these designs so that the implementation is generated systematically from its description in a logic-based specification language. Systematic logic-based transformations [PP99] can be

used to obtain more efficient algorithms and better resource usage. Such transformations can also lead to programs that are more optimizable, as is needed for the update analysis given here to be effective. The target of the transformation can then be subjected to static analysis (e.g., update analysis) to support the generation of efficient, possibly destructive, code in a programming language. The target programming language, in our case, Common Lisp, bridges the gap between the logic-based program and the target machine. The programming language compiler then handles the machine-specific optimizations.

There is a great deal of work to be done before such code generation from a specification language can match or surpass the efficiency of custom code written in a low-level programming language like C. Most of the remaining work is in implementing other forms of analyses similar to the update analysis described in this paper. The C language allows various low-level manipulations on pointers and registers that cannot be emulated in a high-level language, but the vast majority of programs do not exploit such coding tricks. For these programs, we believe that it is quite feasible to generate code from a high-level logic-based specification language that matches the efficiency of the best corresponding C programs.

References

[BH87] A. Bloss and P. Hudak. Path semantics. In *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, number 298 in Lecture Notes in Computer Science, pages 476–489. Springer-Verlag, 1987.

[Blo94] Adrienne Bloss. Path analysis and the optimization of nonstrict functional languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, 1994.

[DP93] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, September 1993.

[Fel90] M. Felleisen. On the expressive power of programming languages. In *European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 35–75. Springer-Verlag, 1990.

[GH89] K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *16th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1989.

[GP98] Deepak Goyal and Robert Paige. A new solution to the hidden copy problem. In *Static Analysis Symposium*, pages 327–348, 1998.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd., 1987. Preliminary version appeared in Proceedings of 1986 ACM Conference on LISP and Functional Programming, August 1986, pages 351–363.

[Ode91] Martin Odersky. How to make destructive updates less destructive. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 25–26, January 1991.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[OS97] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[Pet78] A. Pettorossi. Improving memory utilization in transforming recursive programs. In J. Winkowski, editor, *Proceedings of MFCS 1978*, pages 416–425, Berlin, Germany, 1978. Springer-Verlag.

[PP99] Alberto Pettorossi and Maurizio Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2-3):197–230, 1999.

[SCA93] A. V. S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275, New York, 1993. ACM Press.

[Sch82] Jerald Schwarz. Using annotation to make recursion equations behave. *IEEE Transactions on Software Engineering*, 8(1):21–33, 1982.

[vG96] John H. G. van Groningen. The implementation and efficiency of arrays in Clean 1.1. In *Proc. 8th International Workshop on Implementation of Functional Languages, IFL'96*, number 1268 in Lecture Notes in Computer Science, pages 105–124. Springer-Verlag, 1996.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North-Holland, Amsterdam, 1990.

[Wad97] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193. IEEE, April 1998.