Copy Elimination in Functional Languages

K.Gopinath and John L.Hennessy
Computer Systems Lab
CIS 034, Stanford University, CA 94305
gopi@sonoma.stanford.edu jlh@vsop.stanford.edu

Abstract

Copy elimination is an important optimization for compiling functional languages. Copies arise because these languages lack the concepts of state and variable; hence updating an object involves a copy in a naive implementation. Copies are also possible if proper targeting has not been carried out inside functions and across function calls. Targeting is the proper selection of a storage area for evaluating an expression. By abstracting a collection of functions by a target operator, we compute targets of function bodies that can then be used to define an optimized interpreter to eliminate copies due to updates and copies across function calls. The language we consider is typed lambda calculus with higher-order functions and special constructs for array operations. Our approach can eliminate copies in divide and conquer problems like quicksort and bitonic sort that previous approaches could not handle.

We also present some results of implementing a compiler for a single assignment language called SAL on some small but tough programs. Our results indicate that it is possible to approach a performance comparable to imperative languages like Pascal.

1 Introduction

Copy elimination is an important optimization for implementing functional languages. Though it is related to the problem of copy propagation that has been considered in many compilers [1], the term is used in a more general context where structured values can be updated, copies can be eliminated across function calls and the computation tree can be reordered. Because of these additional possibilities, copy elimination is a hard problem, being undecidable in general.

To efficiently implement functional languages on conventional machines, three main optimizations are required: conversion, where possible, of lazy/call-by-need into call-by-value evaluation; call-by-value into call-by-reference evaluation; and doing updates in-place. Our work concerns itself with the copy elimination that results from the last two parts.

[0]This work was supported in part by NSF grant CCR 8351269.

Using abstract interpretation, a technique that was pioneered by Cousot and Cousot[12] for deriving properties of programs, Mycroft[10] considered the problem of detecting when a call-by-need argument can be turned into a call-by-value argument in the interests of efficiency. Hudak[7] has also used the technique of abstract interpretation successfully to detect updates that can be done in-place by reference counting when call-by-value arguments are used. However, his approach does not extend naturally to divide and conquer problems.

In our approach, we compute the *target* of an expression, namely the proper location where the expression should be evaluated, so that the number of intermediate copies is reduced. This requires that all subexpressions also be properly targeted. We also subsume the problem of updating in-place to one of computing the target of an update statement. We have adapted the term *targeting* from code generation work where targeting is used to avoid unnecessary moves between registers by evaluating some values in certain registers. A good example is the targeting of multiplicand and multiplier into odd and even registers in certain architectures.

Our approach can eliminate copies in divide and conquer problems like quicksort and bitonic sort and achieve a performance comparable to imperative languages like Pascal. We present some results of running a compiler for a single assignment language called SAL on a set of small but tough benchmarks in Section 12. The theory can handle higher-order functions but these are not present in SAL. Call-by-value evaluation will be assumed but it is possible to handle other evaluation mechanisms by changing some definitions in Section 7. We first introduce targeting and the language considered in the theory, followed by domains and the semantics of target expressions and mention some theoretical results that are useful in copy elimination. Some examples of computing targets are then presented. Next, we develop two interpreters, one standard and one using target information and present results on their extensional equivalence. Finally, experimental results are presented.

2 Targeting

Targeting is the proper selection of a storage area for evaluating an expression. A good selection of targets reduces the number of intermediate copies since temporaries may not be

needed. Previous work[8, 7] on copy elimination has concentrated on doing updates in-place but this is not enough in tackling divide and conquer problems. Consider the following simple divide and conquer schema (where **cat** is the array catenate operation):

$$\begin{array}{l}\textbf{type } arr(h) = \textbf{array}[1..h] \textbf{ of } T \\ \textbf{function } f(A : arr(h)) : arr(h) = \\ \quad \textbf{if } h = 1 \textbf{ then } g(A) \\ \quad \textbf{else cat}(f(A[1..h/2]), f(A[h/2+1..h]))\end{array}$$

There are no explicit updates here and an efficient compilation is possible only if a compiler can detect that the divide phase creates two non-overlapping subarrays that are updated across function boundaries without interference from each other. This is achieved if we discover that the body of the function can be targeted to the parameter A. This in turn requires that $g(A)$ be targeted to A, $f(A[1..h/2])$ to $A[1..h/2]$ and $f(A[h/2+1..h])$ to $A[h/2+1..h]$. The first condition is true since the size of array is one (*i.e.* a scalar) and the type of the result is also a scalar. The last two conditions are consistent with the targeting of the function f to its argument A. One can then prove that the result of the function f is given by some sequence of updates on its argument A. We now can proceed to change the value parameter A into a reference parameter after making sure it is safe to do so by checking that no actual parameter that corresponds to A outside of f is live after it is passed to f. This enables us to approach the efficiency of imperative languages for the class of divide and conquer problems.

We follow the following procedure for copy elimination: First, we compute the targets of functions with the liveness analysis being strictly intra-procedural. Fixpoint iteration is needed in general but sometimes type information can be used to guess fixpoints. Next, we convert call-by-value array parameters to call-by-reference parameters if it is safe. This requires inter-procedural analysis. Finally, each expression in the program is decorated with two targets: synthetic and inherited and this can be used to decide when it is necessary to allocate new storage for array-creating expressions and when they can share storage already allocated.

3 Syntax of Types and Expressions

The language we consider is typed lambda calculus with the following special constructs for arrays:

- create (make) array: **mka**(*lb, ub*) creates an array whose lower and upper integer bounds are given by *lb, ub* and each element is undefined.
- update: **upd**(A,i,v), where A is an array, i is a legal index into A and v is of the same type as A's elements, yields an array that is the same as the array A except the ith element is v.
- subarray: **sba**(A,i,j), where A is an array and i and j are legal indices into A, yields the subarray of A starting at index i and ending at j. This will be abbreviated to $A[i..j]$
- catenate: **cat**(A,B), where A and B are arrays, yields the catenation of A and B. For purposes of optimization and the semantics that will be shortly developed, it has the following meaning: if A=$a[l..m]$, B=$a[n..p]$, m+1=n and a (strictly, $a[l..p]$) is not live, then **cat**(A,B) is $a[l..p]$; otherwise, a newly created array which is the catenation of A and B. (Liveness has to be taken into account because A and B could be function calls that return updated versions of an array a in-place when optimization is done.)

We require that an array or a subarray have bounds of a restricted form so that its type can be determined. Otherwise, looping computations on the bounds are possible and a subarray or an array may not be typable. The syntax is as follows:

$$\begin{array}{lcl} t & = & \textbf{int} \mid \textbf{bool} \mid t_1 \rightarrow t_2 \mid \textbf{array}[I_1..I_2] \textbf{ of } t \\ & & \\ & & \text{Most general form of } I_1 \text{ and } I_2 \text{ is } c + \sum_{i=1}(a_i/b_i)z_i \\ & & \text{where } a_i, b_i, c \text{ are integer constants with } b_i > 0 \text{ and} \\ & & z_i \text{ are integer identifiers} \\ & & \\ e & = & c \mid x \mid \lambda x : t.e \mid e_1(e_2) \mid \textbf{if}(e_1, e_2, e_3) \mid \\ & & \textbf{upd}(A, i, v) \mid \textbf{mka}(lb, ub) \mid \textbf{sba}(A, i, j) \mid \textbf{cat}(A, B) \mid \\ & & \textbf{letrec} \\ & & \quad f_1 = \lambda x : t_1.e_1 \\ & & \quad \vdots \\ & & \quad f_n = \lambda x : t_n.e_n \\ & & \textbf{in} \\ & & \quad e \end{array}$$

4 Assumptions and Notation

- We assume that every occurrence of an identifier of array type is made unique. The kth occurrence of x_i in function f is denoted by x_{ik}. We adopt the convention that if an identifier has two subscripts, then we are referring to a particular occurrence of an identifier whereas if it has none or one, we are referring to the identifier itself. The main program e in the **letrec** is considered to be the function f_0.
- Let $lb_x..ub_x$ be the subscript range of an array x; assume that $x[lb_x..ub_x]$ is always rewritten as x.
- When a lambda term is present as an argument to a function, any free variable is changed to a bound variable and the free variable passed as an extra argument to the function. This is similar to lambda-lifting[13] to reduce the number of reductions. We also assume the

following abbreviations: $\lambda x, y.\ e$ for $\lambda x.\lambda y.\ e$ and $f(x,y)$ for $\mathbf{app}(\mathbf{app}(f,x),y)$.

- Let the kth function call of f_i and the functional parameter fp_i in function f_j be f_{ijk} and fp_{ijk} respectively. Let $actual(e,i)$ represent the ith actual parameter of the call statement e (or equivalently, the ith argument of the beta reduction e) and $formal(f_i,j)$ represent the jth formal parameter of f_i (or equivalently, the jth bound variable of f_i). Let $\mathcal{FP}$ be the set of all functional parameters.
- $e[x_1/x_2]$ represents the substitution of all occurrences of x_2 in e by x_1.
- To distinguish between bound variables in the standard and target semantics, we use a dot over the bound variable in the latter case.

5 Domains of Targets

Let $\mathbf{D}$ be the domain of values and let $\perp_D$ be the corresponding bottom value. The domain of targets is $\mathbf{D}_T$ and $\mathcal{T}$ is the semantic function that computes the target of an expression. A first-order target is a set whose elements represent names that could potentially share storage. $\mathbf{D}_T$ has $\perp_T$ (the empty set) and $\top_T$ as the bottom and top elements respectively with $\top_T$ being used as an "error" value. $\wp$ is the powerset symbol. Let $\mathbf{G}$ be a set of unique names for anonymous arrays created by **mka, cat, upd, if, sba** array operations and also for arrays created by applications **app** so that there is a 1-1 correspondence between occurrences of these array creating operations and the set $\mathbf{G}$. These names are statically determined. When discussing an anonymous array expression p, we refer to its unique name by g_p. Let

$$
\begin{array}{rcl}
\mathbf{Exp} & = & \text{domain of expressions as defined in Section 3} \\
\mathbf{Id} & = & \text{domain of array identifiers} \\
\mathbf{A} & = & \mathbf{Id} \cup \mathbf{G} \quad \text{−the domain of array names} \\
\mathbf{S} & = & \{a[l..m] \mid a \in \mathbf{A};\ l,m \in lb_a..ub_a\} - \mathbf{A} \\
 & & \text{−the domain of proper subarrays;} \\
 & & \quad \mathbf{S} \text{ and } \mathbf{A} \text{ are disjoint} \\
\tilde{\mathbf{G}} & = & \wp(\mathbf{G} \cup \text{subarrays of } \mathbf{G}) \\
 & & \text{−domain of all targets derived} \\
 & & \quad \text{from anonymous array expressions} \\
\mathbf{Targets} & = & \wp(\mathbf{A}) \cup \wp(\mathbf{S}) \cup \top_T \\
 & & \text{−the domain of all possible first-order targets} \\
\mathbf{D}_T & = & \mathbf{Targets} + \mathbf{D}_T \rightarrow \mathbf{D}_T \\
 & & \text{−the domain of all possible targets} \\
\mathbf{Env}_T & = & \mathbf{A} \rightarrow \mathbf{D}_T \\
 & & \text{−the domain of targeting environments} \\
\mathcal{T} & : & \mathbf{Exp} \rightarrow \mathbf{Env}_T \rightarrow \mathbf{D}_T \quad \text{−the targeting function}
\end{array}
$$

The lattice structure over domain $\mathbf{D}_T$ is induced by **Targets**. The partial order is defined for target terms of identical type as follows:

$$e_1 \sqsubseteq e_2 \text{ if } \begin{cases} e_1 \subseteq e_2 \text{ if } e_1, e_2 \in \mathbf{Targets} - \tilde{\mathbf{G}} \\ e_1 \in \mathbf{Targets} - \tilde{\mathbf{G}} \text{ and } e_2 \in \tilde{\mathbf{G}} \end{cases}$$

$$\lambda\dot{x}_1.e_1 \sqsubseteq \lambda\dot{x}_2.e_2 = \lambda\dot{x}_1.(e_1 \sqsubseteq e_2[\dot{x}_1/\dot{x}_2]), \quad \dot{x}_1, \dot{x}_2 : t$$

The lub operation is defined for target terms of identical type as follows:

$$e_1 \sqcup e_2 = e_1 \cup e_2 \text{ if } e_1, e_2 \in \mathbf{Targets}$$

$$\lambda\dot{x}_1.e_1 \sqcup \lambda\dot{x}_2.e_2 = \lambda\dot{x}_1.(e_1 \sqcup e_2[\dot{x}_1/\dot{x}_2]), \quad \dot{x}_1, \dot{x}_2 : t$$

We define two monotonic operators α_{if} and α_{cat} (used for defining targets of **if** and **cat** expressions respectively) over the domains $\mathbf{D}_T$ and **Targets** respectively below. If the first two operands are incompatible (*i.e.*, the names represented by these targets cannot share the same storage), the symbolic target for the anonymous array for the **if** or **cat** expression (passed as g) is returned.

$$
\begin{array}{l}
\alpha_{if}(a,b,g) \\
\quad = \quad g \text{ if } \exists aa \subseteq a \text{ and } bb \subseteq b \text{ such that} \\
\qquad aa \in \tilde{\mathbf{G}} \text{ and } bb \in (\mathbf{Targets} - \tilde{\mathbf{G}}) \text{ and vice-versa} \\
\quad = \quad g \text{ if } \exists aa \in a \text{ and } bb \in b \text{ such that } aa = \dot{x}_i[l..m], \\
\qquad bb = \dot{x}_i[n..p], (l \neq n \text{ or } m \neq p), x_i \in \mathbf{A} \\
\quad = \quad g \text{ if } \exists aa \in a \text{ and } bb \in b \text{ such that } aa = \lambda\dot{x}_i.e_1 \\
\qquad \text{and } bb = \lambda\dot{x}_i.e_2 \text{ and } \alpha_{if}(e_1,e_2,g) \in \mathbf{G} \\
\quad = \quad a \sqcup b \text{ otherwise}
\end{array}
$$

The first line considers the incompatible case of a formal and a newly created array on the arms of the **if** statement. There is no *a priori* reason why they should have any storage relationship, so we return the target of the anonymous array corresponding to the **if** statement. The second line takes care of 2 disjoint subarrays of an array. We also adopt the convention in this case and similar cases below that when $x_i \in \mathbf{G}$, then $\dot{x}_i = \{x_i\}$. The third line defines the incompatible cases in the higher-order cases. The last line takes care of all the other cases.

$$
\begin{array}{l}
\alpha_{cat}(a,b,g) \\
\quad = \quad \dot{x}_i[l..p] \text{ if } a = \dot{x}_i[l..m] \text{ and } b = \dot{x}_i[n..p] \\
\qquad \text{and } n = m+1. \\
\quad = \quad \perp_T \text{ if either } a \text{ or } b \text{ is } \perp_T \\
\quad = \quad g \text{ if } \exists aa \in a \text{ and } bb \in b \text{ such that } aa \in \mathbf{A} \\
\qquad \text{and } bb \in \mathbf{S} \text{ and vice-versa} \\
\quad = \quad g \text{ if } \exists aa \subseteq a \text{ and } bb \subseteq b \text{ such that } aa \in \tilde{\mathbf{G}} \\
\qquad \text{and } bb \in (\mathbf{Targets} - \tilde{\mathbf{G}}) \text{ and vice-versa} \\
\quad = \quad g \text{ if } \exists aa \in a \text{ and } bb \in b \text{ such that } aa = \\
\qquad \dot{x}_i[l..m], bb = \dot{x}_i[n..p] \text{ and } n \neq m+1, x_i \in \mathbf{A} \\
\quad = \quad g \text{ if } \exists aa \in a \text{ and } bb \in b \text{ such that } aa = \\
\qquad \dot{x}_i[l..m], bb = \dot{x}_j[n..p], i \neq j \text{ and } x_i, x_j \in \mathbf{A}
\end{array}
$$

The first line considers the case when two arrays that are adjacent can be catenated in-place. The second line is needed

for the fixpoint iteration to proceed properly. The third case takes care of cases like **cat**(A, B[2..6]) that are incompatible since some portions of B get overlaid if in-place catenation is to take place. The next case is similar to the first case in the α_{if}. The next case considers the case of overlapping subarrays of an array. The last case is similar to the third case.

These monotonic operators assume that the arithmetic conditions like $n = m + 1$ are decidable. This is not possible if arbitrary expressions are possible for *l,m,n,p*. Since we are assuming simple linear expressions in the bound variables, it is possible to check the conditions by symbolic analysis. For example, if n is given by $c + \sum_{i=1} a_i z_i$ and m by $d + \sum_{i=1} b_i z_i$, then to check if $n = m+1$, we check if $c - d = 1$ and $a_i = b_i$. However, these conditions do not capture all the cases when $n = m+1$, so the monotonic operators computed by symbolic analysis give a weaker estimate than when integer arithmetic is used. Let $\alpha_{if_{symb}}$ and $\alpha_{cat_{symb}}$ be the monotonic operators when symbolic arithmetic is used. We can show that $\alpha_{if} \sqsubseteq \alpha_{if_{symb}}$ and $\alpha_{cat} \sqsubseteq \alpha_{cat_{symb}}$. Define $\mathcal{T}_{symb} : \mathbf{Exp} \to \mathbf{Env}_T \to \mathbf{D}_T$ as the version of $\mathcal{T}$ where symbolic arithmetic is used for the monotonic operators.

6 Semantics of Target Expressions

Let x_i represent a bound variable, *fp* a functional parameter, *sc* a scalar and *arith-bool* arithmetic and boolean expressions. The function *notlive* is defined in the next section and $\mathcal{F}_{pos}(fp)$, the set of all the possible values for a functional parameter fp, in the appendix. Let $tenv \in \mathbf{Env}_T$. Note that, though each occurrence of an **if, upd, cat**,*etc.* has its own unique name, we refer to the name of the expression p under discussion by the notation g_p.

$\mathcal{T}[\![c]\!]tenv = \top_T$

$\mathcal{T}[\![sc]\!]tenv = \top_T$

$\mathcal{T}[\![arith\text{-}bool]\!]tenv = \top_T$

$\mathcal{T}[\![fp]\!]tenv =$ if the final range of fp is an array then

$$\bigcup_{s \in \mathcal{F}_{pos}(fp)} \mathcal{T}[\![s]\!]tenv \text{ else } \top_T$$

$\mathcal{T}[\![x_{ik}]\!]tenv = \dot{x_{ik}}$

$\mathcal{T}[\![\lambda x_i : t.e]\!]tenv = \lambda \dot{x_i}.\ \mathcal{T}[\![e]\!]tenv[\dot{x_i}/x_i]$

$\mathcal{T}[\![e_1(e_2)]\!]tenv = \mathcal{T}[\![e_1]\!]tenv\ (\mathcal{T}[\![e_2]\!]tenv)$

$\mathcal{T}[\![\mathbf{if}(e_1, e_2, e_3)]\!]tenv = \alpha_{if}(\mathcal{T}[\![e_2]\!]tenv, \mathcal{T}[\![e_3]\!]tenv, \{g_{if}\})$

$\mathcal{T}[\![\mathbf{upd}(A, i, v)]\!]tenv =$

let

$t = \mathcal{T}[\![A]\!]tenv$

in

if $\forall \dot{s} \subseteq t.\ notlive(\dot{s}, \{g_{upd}\})$ then t else $\{g_{upd}\}$

$\mathcal{T}[\![\mathbf{cat}(A, B)]\!]tenv =$

let

$t = \mathcal{T}[\![A]\!]tenv$

in

if $\forall \dot{s} \subseteq t.\ notlive(\dot{s}, \{g_{cat}\})$

then $\alpha_{cat}(t, \mathcal{T}[\![B]\!]tenv, \{g_{cat}\})$ else $\{g_{cat}\}$

$\mathcal{T}[\![\mathbf{mka}(lb, ub)]\!]tenv = \{g_{mka}\}$

$\mathcal{T}[\![\mathbf{sba}(A, i, j)]\!]tenv =$

let

$t = \mathcal{T}[\![A]\!]tenv$

in

$$\bigcup_{\dot{s} \subseteq t} \dot{s}[i..j]$$

$\mathcal{T}[\![\{\mathbf{letrec}\ f_1 = \lambda x_1 : t_1.e_1, \cdots, f_n = \lambda x_n : t_n.e_n\ \mathbf{in}\ e\}]\!]$

$tenv = \mathcal{T}[\![e]\!]tenvv$ where $tenvv =$ least fixed point

$(\lambda tenv.tenv[\cdots, f_i \leftarrow \lambda \dot{x_i}.\mathcal{T}[\![e_i]\!]tenv[\dot{x_i}/x_i], \cdots])$

Even though we have higher-order functions, we do not need "target pairs" (like "alias pairs" or "strictness pairs"[2, 11, 9]) since the target of each of the constructs is the same whether used as a value or function. Most of the above equations are straightforward. The first three cases concern scalars and these do not matter in copy elimination. Hence we return the "error" value. For **upd, cat**, we first have to map A by $\mathcal{T}$ to get the possible (first-order) targets and make sure that identifiers corresponding to them are not live later in some evaluation sequence. Since we are interested in copies due to arrays, a functional parameter that has an array as the final range will have to be applied at some stage to obtain an array (otherwise, we can ignore it); so we find all the possible values it can take and return the targets of these values. This is also needed if we need to take care of **upd**'s of arrays that are generated by applications to a functional parameter. If the expense of computing all the possible values is not desirable, then a simple heuristic is to return the union of all the formal array parameters (in the target semantics) of the functionals that have an array as range instead of returning the value in the *then*-part in the semantics. This, however, reduces the accuracy and copies may not be eliminated.

7 Computing Evaluation Sequences

The computation of targets requires knowledge of the evaluation sequence of an expression since updating in-place depends on determining liveness of a bound variable which in turn depends crucially on the evaluation order. An evaluation sequence is an ordered set of occurrences of identifiers in a function that is encountered during an evaluation. Since each occurrence of an identifier made unique by the use of double-subscripts, liveness information can be obtained from an evaluation sequence. We also record array bounds of the form $[i..j]$ so that liveness of subarrays can also be computed.

Let $\emptyset$ stand for the empty range and $<>$ stand for the empty sequence. Let $::$ be append to a sequence and let $:$ append range information. Let $\rho :: y : \emptyset = \rho :: y$ and $\rho ::< x_i >: [l..m] = \rho ::< x_i[l..m] >$. If σ is a set of sequences, then let $\sigma :: y = \{\rho :: y | \rho \in \sigma\}$. We use the notation $\sigma \backslash x$ to mean eliminating x from all sequences in σ.

Let *BVtoFbody* be a function that maps each identifier to the body of the function where it is bound. The function *occr* computes all the possible set of sequences of a function given some bound variable of the function. The parameter t lets the occurrences of identifiers in the arguments of **cat** to be skipped. This is an optimization based on the way α_{cat} is defined: if A and B are two overlapping subarrays targeted to X (*i.e.* some subarray of X is live when an update in A updates X), then α_{cat} returns g_{cat}. Hence occurrences in A and B can be skipped when the target of **cat** is being computed since the liveness is taken care of in α_{cat}. To be able to do this, *occ* records the beginning and end of the identifiers that occur in the **cat** expression. If t is some $\{g_{cat}\}$, define $\sigma\backslash\backslash t$ as the sequence with all the identifiers between g_{cat_begin} to g_{cat_end} eliminated. Otherwise, it returns σ.

$$
\begin{aligned}
occr(x_i, t) &= occ(BVtoFbody(x_i), \\
&\qquad \{<>\}, \emptyset)\backslash\backslash t \\
occ(c, \sigma, range) &= \sigma \\
occ(fp, \sigma, range) &= \sigma \\
occ(x_{ik}, \sigma, range) &= \sigma ::< x_{ik} : range > \\
occ(e_1(e_2), \sigma, range) &= occ(e_2, occ(e_1, \sigma, \emptyset), range) \\
occ(\mathbf{if}(e_1, e_2, e_3), \sigma, range) &= occ(e_2, occ(e_1, \sigma, \emptyset), range) \cup \\
&\qquad occ(e_3, occ(e_1, \sigma, \emptyset), range) \\
occ(\mathbf{upd}(A, i, v), \sigma, range) &= occ(A, occ(i, occ(v, \sigma, \emptyset), \emptyset), \emptyset) \\
occ(\mathbf{mka}(lb, ub), \sigma, range) &= occ(ub, occ(lb, \sigma, \emptyset), \emptyset) \\
occ(\mathbf{sba}(A, i, j), \sigma, range) &= occ(A, occ(i, occ(j, \sigma, \emptyset), \\
&\qquad \emptyset), [i..j]) \\
occ(\mathbf{cat}(A, B), \sigma, range) &= occ(B, occ(A, \sigma :: g_{cat\_begin}, \\
&\qquad \emptyset), \emptyset) :: g_{cat\_end} \\
occ(\lambda x_i : t.e, \sigma, range) &= \sigma :: (occ(e, \{<>\}, \emptyset)\ \backslash x_i)
\end{aligned}
$$

We are assuming call-by-value evaluation above but by suitably modifying *occr*, we can change the evaluation to call-by-need and other evaluation mechanisms. The above approach is similar to *path semantics* of Bloss and Hudak[3]. The computation of these sequences requires one pass over the functions. The predicate *notlive* checks if there is some occurrence x_{ik} after a given occurrence x_{ij} in some evaluation sequence. In the two sequences below, we omit range information if it is a "don't care".

$$
\begin{aligned}
notlive(\dot{s}, t) = \exists x_{ij}.((\dot{s} = \dot{x}_{ij} \text{ or } \dot{x}_{ij}[l..m]) \text{ and} \\
\neg\exists x_{ik}.\ < \cdots x_{ij} \cdots x_{ik} \cdots > \in occr(x, t)) \text{ or} \\
(\dot{s} = \dot{x}_{ij}[l..m] \text{ and } \neg(\exists x_{ik}[p..q].\ < \cdots x_{ij} \cdots \\
x_{ik}[p..q] \cdots > \in occr(x, t) \text{ and } [l..m] \cap [p..q] \neq \emptyset))
\end{aligned}
$$

The first condition in the *or* considers only whole arrays whereas the second condition considers subarrays.

8 Some Results

Theorem 1 *For any finite program with bounded arrays, the fixpoint tenvv is computable.*

Theorem 2 *Let $e : t_1 \to t_2 \to \cdots \to t_n \to a$, where a is of array type, be rewritten as $\lambda y_1 : t'_1, \cdots, y_l : t'_l.E$ where $l \leq n$, $FreeVariables(E) = \{x_1, \cdots, x_m\}$ and E is of array type. If $\mathcal{T}[\![e]\!]tenv = \lambda y_1, \cdots, y_l.\ \dot{x}_i$ where x_i is of array type and $i \leq m$, then e can be targeted to x_i and x_i updated in-place in e to give the value of e.*

Proof: The proof is by a combination of complete computational induction[16] on the number of arrows and structural induction on E. The proof is given in the appendix.

Corollary 1 *If $\mathcal{T}$ applied to the body of the function f_p is $\dot{x}$ where x is an array bound variable, then the value of the function is given by some sequence of updates on x. Hence x can be converted from a call-by-value parameter to a call-by-reference parameter if all the actuals corresponding to x, in functions other than f_p, are not live after transmission.*

Let *fc* be the set of all possible function calls computed while collecting possible values for functional parameters (See Appendix). If $\dot{x}$ is the target of f_p and x is $formal(f_p, l)$, then x can be converted from a call-by-value parameter to a call-by-reference parameter if

$$
\begin{aligned}
&\forall f_{ijk, \neg(i=j=p)} \in fc. \\
&\quad \forall \dot{s} \subseteq \mathcal{T}[\![actual(f_{ijk}, l)]\!]tenv.\ notlive(\dot{s}, \emptyset)
\end{aligned}
$$

Theorem 3 $\forall e, tenv.\ \mathcal{T}[\![e]\!]tenv\ \sqsubseteq \mathcal{T}_{symb}[\![e]\!]tenv$

9 Examples

9.1 Hudak's Quicksort

We omit type declarations; *v* and *vector* are of array type, others are of type integer. See Hudak[7]. This is a first-order case.

result () = *Quicksort* (*vector*)

Quicksort (*v*) = *qsort* (*v*, 1, *n*)

qsort (*v*, *left*, *right*) =
  if *left* >= *right* **then** *v*
  else *scanright* (*v*, *left*, *right*, *v*[*left*], *left*, *right*)

scanright (*v*, *l*, *r*, *pivot*, *left*, *right*) =
  if *l* = *r* **then** *finish* (**upd**(*v*, *l*, *pivot*), *l*, *left*, *right*)
  else if *v*[*r*] >= *pivot* **then**
    scanright (*v*, *l*, *r*−1, *pivot*, *left*, *right*)
  else
    scanleft (**upd**(*v*, *l*, *v*[*r*]), *l*+1, *r*, *pivot*, *left*, *right*)

scanleft (*v*, *l*, *r*, *pivot*, *left*, *right*) =
  if *l* = *r* **then** *finish* (**upd**(*v*, *l*, *pivot*), *l*, *left*, *right*)
  else if *v*[*l*] <= *pivot* **then**
    scanleft (*v*, *l*+1, *r*, *pivot*, *left*, *right*)
  else
    scanright (**upd**(*v*, *r*, *v*[*l*]), *l*, *r*−1, *pivot*, *left*, *right*)

finish (*v*, *mid*, *left*, *right*) =
  qsort (*qsort* (*v*, *left*, *mid*−1), *mid*+1, *right*)

Fixpoint Iteration The target of each function is computed by iteration (from left to right) in Figure 1. From Theorem 2 and its corollary, we can conclude that quicksort can be implemented in-place.

9.2 Bitonic Sort

(* n is a power of 2; X is a parametric array[1..n] *) This is also a first-order case.

$sb(X)$ = **if** n=1 **then** X **else** *dc* (*trans* (X,1))

rev (X,i) =
  if $i > n/2$ **then** X
  else *rev* (**upd**(**upd**(X,i,$X[n-i+1]$),$n-i+1$,$X[i]$), i+1)

trans (X,i) =
  if $i > n/2$ **then** X
  else if $X[i] \leq X[i+n/2]$ **then** *trans* (X, i+1)
  else *trans* (**upd**(**upd**(X,i,$X[i+n/2]$),$i+n/2$,$X[i]$), i+1)

dc (X) = **cat**($sb(X[1..n/2])$,$sb(X[n/2+1..n])$)

merge (X, Y) = sb(**cat**(X,*rev* (Y)))

sort (X) =
  if n=1 **then** X
  else *merge* (*sort* ($X[1..n/2]$), *sort* ($X[n/2+1..n]$))

The fixpoint iteration is given in Figure 2. The target of *merge* does not converge to a simple value because the nature of arguments X and Y, which are always adjacent, are not taken into account. However, *sort* and other functions converge to simple targets because this information is taken into account. By undertaking a collecting analysis of the arguments of *merge*, the adjacency of the two arguments can be discovered and we can conclude that *merge* also has a simple target.

We get simple targets for *sort* and *sb* only if we assume the evaluation order for **upd** to be from right to left (this can be seen in the definition of *occ*). If this is not the case, *temporary introduction*[6] is needed to eliminate interfering live-ranges.

9.3 Higher-order Example

$g(a,b)=a$
$h(a,b)=b$

$f(a,b,g_1,g_2)=$
  if *cond1* **then**
    if *cond2* **then cat**(**upd**($g_1(a,b)$,i_1,v_1), **upd**(b,i_2,v_2))
    else cat($g_1(a,b)$,**upd**($g_2(a,b)$,i_3,v_3))
  else f(**upd**(a,i_4,v_4),**upd**(b,i_5,v_5),g_1,g_2)

$f(A[1..n/2],A[n/2+1..n],g,h)$

The fixpoint of the function f can be computed to be $\dot{A}$. If the simple heuristic mentioned in Section 6 is used instead of computing all the possible values of the functional parameters, the first update cannot be done in-place and we do not get the target of f to be $\dot{A}$.

The liveness analysis that is inherent in the definitions of *occ* and *notlive* is able to handle the above case satisfactorily; however more detailed liveness analysis has to be undertaken when f is defined instead as follows:

$f(a,b,g_1,g_2)=$
  cat(**upd**($g_1(a,b)$,i_1,v_1), **upd**($g_2(a,b)$,i_2,v_2))

To handle this, *occ* has to know about the targets of the second update. Since our current definition of *occ* does not use the $\mathcal{T}$ function, it is unable to conclude that the fixpoint of this f is also $\dot{A}$.

10 Complexity of Computing Targets

The problem of strictness analysis in the first-order case has been shown by Hudak and Young[9] to be deterministic exponential time in the number of arguments to the function being analyzed. Based on this, Neirynck[11] concludes that the analysis for computing alias and support sets for higher-order imperative languages to be exponential in time. Since computing targets involves a set of equations similar to Neirynck's, it is at least exponential. This is due to use of extensional equality for functional terms in the fixpoint iteration. Further, collecting all the possible values for a functional parameter has been shown by Weihl[5] to be P-space hard.

However, we believe that the average behaviour is much better. Neirynck[11] shows that for the first-order case, type information can be used to guess the correct fixpoint. However, heuristics have to be used and accuracy may be lost in the higher-order case.

In our analysis, the important question is whether the fixpoint is a simple target; otherwise sharing is not possible. The type of the arguments and the type of the result of a function can often be used to guess the fixpoint, which can then be verified. This is the approach followed in our implementation of the compiler for the first-order case in the language SAL. The complexity in this case is linear in the size of the program if we assume linear induction variables. As an example of the effectiveness of this approach, we found that for the program *puzzle*(see Section 12), the frontend consumed approximately 4 secs on a Sun-3 to generate the intermediate graph representation of the program whereas the analysis in the backend took about the same time. The backend includes, in addition to target analysis, loop invariant motion, common subexpression elimination among loop invariants, computing operands of expressions, introduction of

Quicksort	$\lambda\dot{v}.\bot_T$	$\lambda\dot{v}.\bot_T$	$\lambda\dot{v}.\dot{v}$	$\lambda\dot{v}.\dot{v}$	$\lambda\dot{v}.\dot{v}$
qsort	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$
scanright	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$
scanleft	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$
finish	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\bot_T$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$	$\lambda\dot{v},\cdots.\dot{v}$

Figure 1: Fixpoint Iteration in Quicksort

sb	$\lambda\dot{X}.\bot_T$	$\lambda\dot{X}.\dot{X}$	$\lambda\dot{X}.\dot{X}$	$\lambda\dot{X}.\dot{X}$
rev	$\lambda\dot{X},i.\bot_T$	$\lambda\dot{X},i.\dot{X}$	$\lambda\dot{X},i.\dot{X}$	$\lambda\dot{X},i.\dot{X}$
trans	$\lambda\dot{X},i.\bot_T$	$\lambda\dot{X},i.\dot{X}$	$\lambda\dot{X},i.\dot{X}$	$\lambda\dot{X},i.\dot{X}$
dc	$\lambda\dot{X}.\bot_T$	$\lambda\dot{X}.\bot_T$	$\lambda\dot{X}.\dot{X}$	$\lambda\dot{X}.\dot{X}$
merge	$\lambda\dot{X},\dot{Y}.\bot_T$	$\lambda\dot{X},\dot{Y}.\bot_T$	$\lambda\dot{X},\dot{Y}.\alpha_{cat}(\dot{X},\dot{Y},\{g_{cat}\})$	$\lambda\dot{X},\dot{Y}.\alpha_{cat}(\dot{X},\dot{Y},\{g_{cat}\})$
sort	$\lambda\dot{X}.\bot_T$	$\lambda\dot{X}.\dot{X}$	$\lambda\dot{X}.\dot{X}$	$\lambda\dot{X}.\dot{X}$

Figure 2: Fixpoint Iteration in Bitonic sort

temporaries to break dependencies (so that nested updates in-place are possible), some dependency analysis, etc. [6].

11 Equivalence of Standard and Optimized Interpreters

Using the target information, we can define two interpreters, with and without optimization. We can show that they compute the same values (extensional equality) though their use of store and the time taken for execution can be very different. We first define the standard interpreter without any optimization: call this $\mathcal{E}_{std}$. The domains are as follows:

$$
\begin{array}{rcl}
\mathbf{Loc} & = & \mathbf{Integer} \times \mathbf{Integer} \\
 & & -\text{the domain of locations} \\
\mathbf{Env} & = & \mathbf{A} \to \mathbf{D} \quad -\text{the domain of environments} \\
\mathbf{F} & = & \mathbf{D} \to \mathbf{Env} \to \mathbf{St} \to (\mathbf{D} \times \mathbf{Env} \times \mathbf{St}) \\
 & & -\text{the domain of functions} \\
\mathbf{D} & = & \mathbf{Bool} + \mathbf{Nat} + \mathbf{F} + \mathbf{Loc} \\
 & & -\text{the domain of values} \\
\mathbf{St} & = & \mathbf{Loc} \to \mathbf{D} + \underline{unused} \quad -\text{the domain of stores} \\
\mathcal{E}_{std} & : & \mathbf{Exp} \to \mathbf{Env} \to \mathbf{St} \to \mathbf{D} \times \mathbf{Env} \times \mathbf{St} \\
 & & -\text{the unoptimized semantic function} \\
 & & \quad \text{for expressions} \\
\mathcal{E}_{opt} & : & \mathbf{Exp} \to \mathbf{Env} \to \mathbf{St} \to \mathbf{D} \times \mathbf{Env} \times \mathbf{St} \\
 & & -\text{the optimized semantic function} \\
 & & \quad \text{for expressions} \\
\mathcal{K} & : & \mathbf{Con} \to \mathbf{D} \\
 & & -\text{the semantic function for constants} \\
Allocate & : & \mathbf{St} \to Type \to \mathbf{Loc} \times \mathbf{St} \quad -\text{returns a} \\
 & & \quad \text{location of size given by a type} \\
Eval & : & \mathbf{St} \times \mathbf{Loc} \to \mathbf{D} \quad -\text{reads the value} \\
 & & \quad \text{pointed by a location}
\end{array}
$$

Let $\mathcal{E}_{std,v}$, $\mathcal{E}_{std,e}$ and $\mathcal{E}_{std,s}$ stand for the value, environment and store components respectively. $\mathcal{K}$ gives meaning to constants. The standard interpreter does not need the store parameter but it is useful in showing the equivalence of the two interpreters.

A location *loc* has two components $< l, len >$ with l the starting address of the location and *len* the length of the structure. We assume a function *Allocate*, which given a store and a type returns a store and a location of the newly allocated space of size determined by the type. A function *Deallocate* can be similarly defined but to keep the interpreters simple, we ignore details about deallocation. We use a two-level store for structures and a one-level store for other values. To access a structure, we first have to dereference once to get its location and then use it to locate the value. Hence the **D** domain contains **Loc**. Thus to access the value of a structure x, we need to say $st(env(x))$ whereas to access other values y, we need to say $env(y)$. Let *Eval* be a function which given a store and a location returns the value at the location. Let $env[x \mapsto loc]$ represent a new environment that is different from *env* in that x is mapped to location *loc*. The standard optimizer is as follows:

$$
\begin{array}{l}
\mathcal{E}_{std}[\![c]\!]env\ st\ = < \mathcal{K}(c), env, st > \\
\mathcal{E}_{std}[\![fp]\!]env\ st\ = < env(fp), env, st > \\
\mathcal{E}_{std}[\![sc]\!]env\ st\ = < env(sc), env, st > \\
\mathcal{E}_{std}[\![x_i]\!]env\ st\ = < st(env(x)), env, st > \\
\mathcal{E}_{std}[\![\lambda x : t.e]\!]env\ st\ = \\
\quad < \lambda loc.\ \mathcal{E}_{std,v}[\![e]\!]env[x \mapsto loc]\ st, env, st > \\
\mathcal{E}_{std}[\![e_1(e_2)]\!]env\ st\ = \\
\quad \text{let} \\
\qquad < f, env_1, st_1 > = \ \mathcal{E}_{std}[\![e_1]\!]env\ st
\end{array}
$$

$< a, env_2, st_2 > = \mathcal{E}_{std}[\![e_2]\!]env_1\ st$
if $type(g_{app})$ is of array type then
$< loc, st_3 > = Allocate(st_2, type(g_{app}))$
$< l, env_3, st_4 > = (f\ a\ env_2\ st_3)$
$g_{app} \overset{copy}{=} l$
$env_4 = env_3[g_{app} \mapsto loc]$
else $< g_{app}, env_4, st_4 > = (f\ a\ env_2\ st_2)$
in
$< g_{app}, env_4, st_4 >$

$\mathcal{E}_{std}[\![\mathbf{if}(e_1, e_2, e_3)]\!]env\ st\ =$
let
$< bool, env_1, st_1 > = \mathcal{E}_{std}[\![e_1]\!]env\ st$
in
if $bool$ then $\mathcal{E}_{std}[\![e_2]\!]env_1\ st_1$
else $\mathcal{E}_{std}[\![e_3]\!]env_1\ st_1$

$\mathcal{E}_{std}[\![\mathbf{upd}(A, i, v)]\!]env\ st\ =$
let
$< ind, env_1, st_1 > = \mathcal{E}_{std}[\![i]\!]env\ st$
$< val, env_2, st_2 > = \mathcal{E}_{std}[\![v]\!]env_1\ st_1$
$< loc, st_3 > = Allocate(st_2, type(g_{upd}))$
$< l, env_3, st_4 > = \mathcal{E}_{std,v}[\![A]\!]env_2\ st_3$
$g_{upd} \overset{copy}{=} l$
update g_{upd} destructively in the
ind position by val
$env_4 = env_3[g_{upd} \mapsto loc]$
in
$< g_{upd}, env_4, st_4 >$

$\mathcal{E}_{std}[\![\mathbf{cat}(A, B)]\!]env\ st\ =$
let
$< v_1, env_1, st_1 > = \mathcal{E}_{std}[\![A]\!]env\ st$
$< v_2, env_2, st_2 > = \mathcal{E}_{std}[\![B]\!]env_1\ st_1$
$< loc, st_3 > = Allocate(st_2, type(g_{cat}))$
$g_{cat} \overset{copy}{=} v_1||v_2(catenate)$
$env_3 = env_2[g_{cat} \mapsto loc]$
in
$< g_{cat}, env_3, st_3 >$

$\mathcal{E}_{std}[\![\mathbf{mka}(lb, ub)]\!]env\ st\ =$
let
$< m, env_1, st_1 > = \mathcal{E}_{std}[\![lb]\!]env\ st$
$< n, env_2, st_2 > = \mathcal{E}_{std}[\![ub]\!]env_1\ st_1$
$< loc, st_3 > = Allocate(st_2, type(g_{mka}))$
$env_3 = env_2[g_{mka} \mapsto loc]$
in
$< g_{mka}, env_3, st_3 >$

$\mathcal{E}_{std}[\![\mathbf{sba}(A, i, j)]\!]env\ st\ =$
let
$< v, env_1, st_1 > = \mathcal{E}_{std}[\![A]\!]env\ st$
$< m, env_2, st_2 > = \mathcal{E}_{std}[\![i]\!]env_1\ st_1$
$< n, env_3, st_3 > = \mathcal{E}_{std}[\![j]\!]env_2\ st_2$
$< loc, st_4 > = Allocate(st_3, type(g_{sba}))$
$g_{sba} \overset{copy}{=} v[m..n]$
$env_4 = env_3[g_{sba} \mapsto loc]$
in
$< g_{sba}, env_4, st_4 >$

$\mathcal{E}_{std}[\![\{\mathbf{letrec}\ f_1 = \lambda x : t_1.e_1, \cdots,$
$f_n = \lambda x : t_n.e_n\ \mathbf{in}\ e\}]\!]env\ st\ = \mathcal{E}_{std}[\![e]\!]envv\ st$
where $envv =$ least fixed point
$(\lambda env.env[\cdots, f_i \leftarrow \mathcal{E}_{std}[\![\lambda x : t_i.e_i]\!]env\ st, \cdots])$

Next we define an optimized interpreter, $\mathcal{E}_{opt}$, by the following steps:

- For each function f_i returning an array type, we operate $\mathcal{T}_{symb}$ on the function body. If the resulting value is $\dot{x}$ where x is an aggregate bound variable, then no storage is allocated to the result of the function since it can share storage with x.
- We decorate the program with two targets at each node of array type: the synthetic and inherited — the synthetic target is computed from the node whereas the inherited one is the target that is passed from the parent node.
- If the synthetic and inherited targets of an expression of array type coincide then the expression is properly targeted and the expression can be evaluated in the storage corresponding to the target. If targets do not coincide, then the expression is evaluated in the storage corresponding to the synthetic target and then copied into the storage corresponding to the inherited target.
- A final step in the optimization is the conversion of a call-by-value parameter into call-by-reference by checking that all the actuals corresponding to the formal, occurring outside of f_i, are not live. However, we will not consider this additional complication in $\mathcal{E}_{opt}$.

Both interpreters are the same on computations not involving arrays. In the description below we will use the notation $(T_i, T_s).exp$ to represent the decoration of a node with the two targets. We often use the simpler notation $T_i.exp$ where T_i is the inherited target (since T_s can always be calculated from the expression). We omit it completely in some cases where it is not necessary, for example, where scalar values are being computed. The notation $(-, T)$ means that only the synthetic target is available.

As an example, we present a decorated program for the simple divide and conquer schema considered earlier:

function $f(A : arr(h)) : arr(h) =$

$$(\dot{A},\dot{A}).\textbf{if } h=1 \textbf{ then } \dot{A}.g(A)$$
$$(\dot{A},\dot{A}).\textbf{else } (\dot{A},\dot{A}).\textbf{cat}(f(\textbf{sba}(A,1,h')),$$
$$f(\textbf{sba}(A,h'+1,h)))$$

Let *Elem* be a function that maps a bound variable in the target semantics to its corresponding bound variable in the standard semantics. The optimized interpreter is as follows:

$\mathcal{E}_{opt}[\![c]\!]env\ st\ = < \mathcal{K}(c), env, st >$
$\mathcal{E}_{opt}[\![fp]\!]env\ st\ = < env(fp), env, st >$
$\mathcal{E}_{opt}[\![sc]\!]env\ st\ = < env(sc), env, st >$
$\mathcal{E}_{opt}[\![(T,\dot{x_i}).x_i]\!]env\ st\ =$
  let
    if $T = \dot{x_i}$ then *noop*
    else $Elem(T) \stackrel{copy}{=} x$
  in
    $< Elem(T), env, st >$
$\mathcal{E}_{opt}[\![\lambda x : t.e]\!]env\ st\ =$
  $< \lambda loc.\ \mathcal{E}_{opt,v}[\![(T_s,T_s).e]\!]env[x \mapsto loc]\ st, env, st >$
$\mathcal{E}_{opt}[\![(T_i,T_s).e_1(e_2)]\!]env\ st\ =$
  let
    $< f, env_1, st_1 > = \mathcal{E}_{opt}[\![(-,T_1).e_1]\!]env\ st$
    $< a, env_2, st_2 > = \mathcal{E}_{opt}[\![(-,T_2).e_2]\!]env_1\ st_1$
    if $type(g_{app})$ is of array type then
      if $T_i = T_s$ and $T_i \neq \{g_{app}\}$ then
        Assign $loc = env(Elem(T_i))$
        as the location for the result of f.
        $< v, env_3, st_4 > = (f\ a\ env_2\ st_2)$
        $env_4 = env_3[g_{app} \mapsto loc, Elem(T_i) \mapsto \perp_D]$
      else
        $< loc, st_3 > = Allocate(st_2, type(g_{app}))$
        $< l, env_3, st_4 > = (f\ a\ env_2\ st_3)$
        $g_{app} \stackrel{copy}{=} l$
        $env_4 = env_3[g_{app} \mapsto loc]$
    else $< g_{app}, env_4, st_4 > = (f\ a\ env_2\ st_2)$
  in
    $< g_{app}, env_4, st_4 >$
$\mathcal{E}_{opt}[\![T.\textbf{if}(e_1,e_2,e_3)]\!]env\ st\ =$
  let
    $< bool, env_1, st_1 > = \mathcal{E}_{opt}[\![e_1]\!]env\ st$
  in
    if *bool* then $\mathcal{E}_{opt}[\![T.e_2]\!]env_1\ st_1$
    else $\mathcal{E}_{opt}[\![T.e_3]\!]env_1\ st_1$
$\mathcal{E}_{opt}[\![T_i.\textbf{upd}((-,T_s).A,i,v)]\!]env\ st\ =$
  let
    $< ind, env_1, st_1 > = \mathcal{E}_{opt}[\![i]\!]env\ st$
    $< val, env_2, st_2 > = \mathcal{E}_{opt}[\![v]\!]env_1\ st_1$
    if $T_i = T_s$ then
      $< v, env_3, st_4 > = \mathcal{E}_{opt}[\![T_s.A]\!]env_2\ st_3$
      $loc = env_3(v)$
      $env_4 = env_3[g_{upd} \mapsto loc, Elem(T_s) \mapsto \perp_D]$
    else
      $< loc, st_3 > = Allocate(st_2, type(g_{upd}))$
      $< l, env_3, st_4 > = \mathcal{E}_{opt}[\![A]\!]env_2\ st_3$
      $g_{upd} \stackrel{copy}{=} l$
      $env_4 = env_3[g_{upd} \mapsto loc]$
    update g_{upd} destructively in the
    ind position by *val*
  in
    $< g_{upd}, env_4, st_4 >$
$\mathcal{E}_{opt}[\![(T_i,T_s).\textbf{cat}(A,B)]\!]env\ st\ =$
  let
    Let l, m be the bounds of the array A
    and $p,\ q$ that of B.
    if $T_i = T_s$ and $T_s \neq \{g_{cat}\}$ then
      $< v_1, env_1, st_1 > = \mathcal{E}_{opt}[\![T_i[l..m].A]\!]env\ st$
      $< v_2, env_2, st_2 > =$
        $\mathcal{E}_{opt}[\![T_i[m+1..q].B]\!]env_1\ st_1$
      $st_3 = st_2;\ loc = < (env(v_1))_1, q-l+1 >$
    else
      $< v_1, env_1, st_1 > = \mathcal{E}_{opt}[\![A]\!]env\ st$
      $< v_2, env_2, st_2 > = \mathcal{E}_{opt}[\![B]\!]env_1\ st_1$
      $< loc, st_3 > = Allocate(st_2, type(g_{cat}))$
      $g_{cat} \stackrel{copy}{=} v_1||v_2$
    $env_3 = env_2[g_{cat} \mapsto loc]$
  in
    $< g_{cat}, env_3, st_3 >$
$\mathcal{E}_{opt}[\![\textbf{mka}(lb,ub)]\!]env\ st\ =$
  let
    $< m, env_1, st_1 > = \mathcal{E}_{opt}[\![lb]\!]env\ st$
    $< n, env_2, st_2 > = \mathcal{E}_{opt}[\![ub]\!]env_1\ st_1$
    $< loc, st_3 > = Allocate(st_2, type(g_{mka}))$
    $env_3 = env_2[g_{mka} \mapsto loc]$
  in
    $< g_{mka}, env_3, st_3 >$
$\mathcal{E}_{opt}[\![(T_i,-).\textbf{sba}((-,T_s).A,i,j)]\!]env\ st\ =$
  let
    $< v, env_1, st_1 > = \mathcal{E}_{opt}[\![(-,T_s).A]\!]env\ st$
    $< m, env_2, st_2 > = \mathcal{E}_{opt}[\![i]\!]env_1\ st_1$
    $< n, env_3, st_3 > = \mathcal{E}_{opt}[\![j]\!]env_2\ st_2$
    if $T_i = \dot{x}[m..n]$ and $T_s = \dot{x}$ then
      $st_4 = st_3$
    $loc = < (env(v))_1 + m - l, n - m + 1 >$

where l is the lower range of the array g_{sba}

else

$< loc, st_4 > = Allocate(st_3, type(g_{sba}))$

$g_{sba} \stackrel{copy}{=} v[m..n]$

$env_4 = env_3[g_{sba} \mapsto loc]$

in

$< g_{sba}, env_4, st_4 >$

$\mathcal{E}_{opt}[\![\{\textbf{letrec } f_1 = \lambda x : t_1.e_1, \cdots, f_n = \lambda x : t_n.e_n$
$\textbf{in } e\}]\!]env\ st\ =\ \mathcal{E}_{opt}[\![(-, T_s).e]\!]envv\ st$
where $envv$ = least fixed point
$(\lambda env.env[\cdots, f_i \leftarrow \mathcal{E}_{opt}[\![\lambda x : t_i.e_i]\!]env\ st, \cdots])$

When a structure is updated destructively to represent a new name, the value of the previous name mapped to this structure is set to $\perp_D$ as a tag. This can be seen in the **upd** case where env_4 is set to a new value. In the unoptimized interpreter, new structures are created for names that could potentially share storage in the optimized version; hence, the optimized version has a fewer number of names defined at any time than the unoptimized version.

Definition 1 *Two pairs of environments and stores* $< env_{std}, st_{std} >$, $< env_{opt}, st_{opt} >$ *correspond* ($\approx$) *if*

- $\forall x.env_{opt}(x) \sqsubseteq env_{std}(x)$
- *If* $env_{opt}(x) <> \perp_D$ *then* $Eval(env_{std}(x), st_{std}) = Eval(env_{opt}(x), st_{opt})$

Theorem 4

$\forall$terminating expressions e of array type,
two environments and stores that correspond
$< env_{std}, st_{std} > \approx < env_{opt}, st_{opt} >$.
$\mathcal{E}_{opt,v}[\![(T_i, T_s).e]\!]env_{opt}\ st_{opt}\ =\ \mathcal{E}_{std,v}[\![e]\!]env_{std}\ st_{std}$
$\mathcal{E}_{opt,es}[\![(T_i, T_s).e]\!]env_{opt}\ st_{opt}\ \approx\ \mathcal{E}_{std,es}[\![e]\!]env_{std}\ st_{std}$

12 Experimental Results

A compiler for a substantial part of SAL has been implemented to verify the effectiveness of the approach. SAL is a single assignment language defined at Stanford[14] and provides iteration, parametric types and streams with scoping mechanisms similar to Algol languages but does not have higher-order functions. It has many features that are found in comparable languages like VAL and SISAL.

The timings for the following programs on a MicroVax-II (without counting the output times except when negligible) were collected using the UNIX *time* command (Figure 3)

- **Quicksort**: sorts an array of 1000 elements.
- **Bitonic sort**: sorts an array of 1024 elements (has to be a power of 2)
- **Bubblesort**: sorts an array of 1000 elements (same as quicksort).
- **Life program**: 500 iterations on a board 10 by 10 (with border 12 by 12)
- **Matrix multiply**: of two 100 by 100 integer matrices
- **8 queens**: Finds all the 92 solutions.
- **NEWRZ**: a time-critical routine from SIMPLE that is used for hydrodynamic calculations. It is a transliteration of the NEWRZ program considered by Ellis[4]
- **CYK**: the Cocke-Younger-Kasami algorithm parses an input string of 128 *a*'s for the following ambiguous grammar: $A \rightarrow a \qquad A \rightarrow AA$
- **Puzzle**: finds the solution to a three-dimensional puzzle. This is a highly recursive and computationally demanding program that is often used for benchmarking C and other languages on workstations.
- **Perm**: enumerates permutations of an array of 7 elements. This is iterated 5 times.

The various optimization levels are: *NoOpt* (No optimization was done); *Opt1* (All optimizations with no rangecheck elimination); *Opt2* (All optimizations with rangecheck elimination by analysis); *Opt3* (All optimizations with rangechecking turned off) whereas *pc* is execution time for Berkeley Pascal with rangechecking turned off. *NoOpt/Opt1* measures the improvement due to high-level optimizations and *Opt3/pc* compares SAL and *pc* timings. There is substantial possibility for improvement in the execution times by use of peephole optimization, register allocation and other standard optimizations. We believe that the timings could be improved by as much as 50% with an UCODE (the intermediate code generated) to UCODE optimizer incorporating these standard optimizations.

It is quite encouraging that we report execution times for five out of ten programs better than or the same as the timings for *pc*. The timings for *life* and *cyk* suffer because of the inability to redefine arrays partially in a straightforwardly efficient way in SAL. There are no "copies" that can be eliminated in 8 *queens* (a backtracking algorithm) and *life*. The matrix multiply program suffers because the copy elimination comes at the cost of removing a loop invariant that is present in the source in a tight loop. However, an UCODE-UCODE optimizer would have reintroduced this loop invariant. All copies in other programs are eliminated by the compiler. The compiler converts the $\mathcal{O}(n^3)$ bubblesort algorithm (due to copies) back to $\mathcal{O}(n^2)$. Similar improvements are seen in *puzzle* and *newrz*. However, *quicksort* and *bitonic sort* do not show such improvement since the copies that are generated become smaller as the recursion unfolds.

	NoOpt	*Opt1*	*Opt2*	*Opt3*	*pc*	*NoOpt/Opt1*	*Opt3/pc*
quick	12.5	2.8	2.8	1.5	1.8	4.93	.83
bitonic	14.3	2.9	2.9	2.5	2.0	4.46	1.25
bubble	1913.2	26.6	17.5	17.5	22.4	71.90	.78
life	23.4	22.6	18.4	18.4	8.6	1.04	2.14
mm	72.6	82.2	48.2	48.2	39.2	0.88	1.23
8queens	1.3	1.3	1.2	1.0	1.2	1.00	.83
newrz	19.9	1.6	1.6	1.2	1.3	12.44	.92
cyk	58.0	56.9	41.9	39.0	18.8	1.02	2.07
puzz	393.6	32.6	30.7	24.0	19.5	12.07	1.27
perm	5.5	3.5	3.5	2.5	2.5	1.57	1.00

Figure 3: Execution times of benchmarks in SAL and Berkeley Pascal.

13 Conclusions

We have discussed a way for removing copies even in the presence of subarrays that are dynamically created and composed. Since this happens very naturally in divide and conquer problems, much of the analysis can be avoided if this idiom can be incorporated in the language by a special construct. This actually has been proposed by some recent work at Yale University[15]. It remains to be seen whether any other such constructs are needed to avoid expensive analysis for copy elimination.

14 Acknowledgements

We would like to thank Paul Hudak, Adrienne Bloss, Daniel Weise and Carolyn Talcott for their many suggestions for improving the earlier versions of this paper. Our thanks are also due to P. Panangaden and Anne Neirynck. We also thank Steve Tjiang, N. Shankar, Thomas Pressburger, Steve Richardson, M. Ganapathi and Jeffrey Barth for their helpful comments on the later versions of the paper. This work was supported in part by NSF grant CCR 8351269; this support is gratefully acknowledged.

Appendix: **Collecting possible values for functional parameters**

- Let $\mathcal{FC}$ be the set of all function calls in the program except those calls on functional parameters.
- Let fv be the set of functions visited, and fc the set of possible function calls.
- Let $\mathcal{C}(x, fv, fc)$ be the set of actual parameters of the formal parameter x when fv function nodes have already been visited and fc is the set of all function calls that has been computed so far.

$\mathcal{ALL}(fc)$ computes the set of all possible values for each functional parameter and the computation is started by the call $\mathcal{ALL}(\mathcal{FC})$.

If $formal(f_i, l)$ is x, then

$$\mathcal{C}(x, fv, fc) = \bigcup \begin{cases} \forall f_{ijk} \in fc. \\ \quad \text{if } actual(f_{ijk}, l) = formal(f_j, m) \text{ then} \\ \quad\quad \text{if } f_j \in fv \text{ then } \emptyset \\ \quad\quad \text{else } \mathcal{C}(formal(f_j, m), \{f_j\} \cup fv, fc) \\ \quad \text{else } \{actual(f_{ijk}, l)\} \\ \forall fp_{ijk} \in fc.\ \mathcal{F}_{pos}(\mathcal{FP}_i) \end{cases}$$

Let $\mathcal{F}_{pos}(\mathcal{FP}_i) = \emptyset, \quad i = 1..|\mathcal{FP}|$

$\mathcal{ALL}(fc) =$

iterate till no change in fc

$$\begin{cases} \mathcal{F}_{pos}(\mathcal{FP}_i) \cup = \mathcal{C}(\mathcal{FP}_i, \emptyset, fc), \quad i = 1..|\mathcal{FP}| \\ fc \cup = \bigcup_{i=1}^{i=|\mathcal{FP}|} \mathcal{F}_{pos}(\mathcal{FP}_i) \\ \{f_{ijk}\} \cup = \mathcal{F}_{pos}(\mathcal{FP}_i) \text{ if } \mathcal{FP}_i \text{ is called in } f_j. \end{cases}$$

Appendix: **Proof of Theorem 2**

The proof is by a combination of complete computational induction[16] on the number of arrows and structural induction on E. To prove the base case $n = 0$, we use structural induction on E as follows:

x_i: The result is immediate.

$e_1(e_2)$: The target is $\dot{x_i}$ only if the target of e_1 maps the target of the argument e_2 into $\dot{x_i}$. There are two possibilities: either $\mathcal{T}[\![e_1]\!]tenv = \lambda\dot{q}.\dot{q}$ and $\mathcal{T}[\![e_2]\!]tenv = \dot{x_i}$ or $\mathcal{T}[\![e_1]\!]tenv = \lambda\dot{q}.\dot{x_i}$ and $\mathcal{T}[\![e_2]\!]tenv = \dot{p}$. In the first case, by induction hypothesis, e_2 is given by updates on x_i and hence $e_1(e_2)$ will also be given by updates on x_i. In the second case, $e_1 = \lambda q.$(expression in $x_i, \cdots$). The expression is therefore given by updates on x_i by induction hypothesis, hence $e_1(e_2)$ is also given by updates on x_i.

if(*cond*,*conseq*,*alt*): The target is $\dot{x_i}$ only if both arms of the conditional map to the same target. From the structural induction hypothesis, it follows that the result is given by the update of the parameter x_i if *cond* is true and also by x_i when *cond* is false. Hence the result is given by update of x_i.

cat($A[l..m]$,$B[n..p]$): The target is $\dot{x_i}$ only if both A and B are mapped to $\dot{x_i}$ and $n{=}m{+}1$ with l and p as the lower and upper bounds of x_i. Hence the result of E is given by update of x_i if **cat** function is in-place for this condition.

upd(A,ii,v): The target is $\dot{x_i}$ only if A is mapped to x_i. If A is not live then x_i can be updated to give the result of f_i.

sba(A,ii,jj): This case gives rise to $\dot{x_i}$ only if i and j are the lower and upper bounds of x_i and A's target is $\dot{x_i}$. Under these conditions, from the inductive hypothesis, if A is given by an update of x_i, then the same update of x_i can give rise to the value of **sba** expression.

$\lambda x_k : t.e$, *c, p, arith-bool*, **mka**: These cases cannot give rise to $\dot{x_i}$ as a target, hence vacuously true.

This completes the proof of the base case.

Assume that the theorem is true for n. To prove for $n+1$, we proceed as follows: Let e have n arrows in its type. Then $e' = \lambda y_0 : t'_0.e$ has $n+1$ arrows in its type. Now $\mathcal{T}[\![e']\!]tenv = \lambda \dot{y_0}.\mathcal{T}[\![e]\!]tenv = \lambda \dot{y_0}.\lambda \dot{y_1}, \cdots, \dot{y_l}.\ \dot{x_i} = \lambda \dot{y_0}, \cdots, \dot{y_l}.\ \dot{x_i}$. Since e can be given by updates on x_i, it follows that e' can also be given by updates on x_i.

This completes the theorem's proof. □

References

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1977.

[2] A.Neirynck, P.Panangaden, and A.J.Demers. Computation of aliases and support sets. In *ACM Symposium on Principles of Programming Languages*, ACM, Jan 1987.

[3] Adrienne Bloss and Paul Hudak. Path semantics. In *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, Springer-Verlag, 1988.

[4] J.R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, December 1984.

[5] William E.Weihl. Interprocedural analysis in the presence of pointers, procedure variables, and label variables. In *ACM Symposium on Principles of Programming Languages*, ACM, Jan 1980.

[6] K. Gopinath. *Copy Elimination in Single Assignment Languages*. PhD thesis, Stanford University, Mar 1988.

[7] Paul Hudak. A semantic model of reference counting and its abstraction. In *ACM Symposium on Lisp and Functional Programming*, ACM, Aug 1986.

[8] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *ACM Symposium on Principles of Programming Languages*, ACM, Jan 1985.

[9] Paul Hudak and Jonathan Young. High-order strictness analysis in untyped lambda calculus. In *ACM Symposium on Principles of Programming Languages*, ACM, Jan 1986.

[10] Alan Mycroft. *Abstract interpretation and Optimising transformations for applicative programs*. PhD thesis, Edinburgh University, 1981.

[11] Anne Neirynck. *Static Analysis of Aliases and Side Effects in Higher-Order languages*. PhD thesis, Cornell University, February 1988.

[12] P.Cousot and R.Cousot. Abstract interpretation. In *ACM Symposium on Principles of Programming Languages*, ACM, Jan 1977.

[13] R.J.M.Hughes. *Graph Reduction with Super-combinators*. Technical Report, Oxford University PRG Technical Monograph PRG-28, 1982.

[14] J.R.Celoni S.J. and J.L.Hennessy. *SAL: A Single-Assignment Language for Parallel Algorithms*. Technical Report, Computer Systems Laboratory, Stanford University, July 1981.

[15] Z.G.Mou and P.Hudak. An algebraic model for divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2(3), 1988.

[16] Z.Manna. *Mathematical theory of computation*. McGraw-Hill, 1974.