# Efficiently Implementing the Copy Semantics of MATLAB's Arrays in JavaScript

Vincent Foley-Bourgon
McGill University, Canada
vincent.foley-bourgon@mail.mcgill.ca

Laurie Hendren
McGill University, Canada
hendren@cs.mcgill.ca

## Abstract

Compiling MATLAB—a dynamic, array-based language—to JavaScript is an attractive proposal: the output code can be deployed on a platform used by billions and can leverage the countless hours that have gone into making JavaScript JIT engines fast. But before that can happen, the original MATLAB code must be properly translated, making sure to bridge the semantic gaps of the two languages.

An important area where MATLAB and JavaScript differ is in their handling of arrays: for example, in MATLAB, arrays are one-indexed and writing at an index beyond the end of an array extends it; in JavaScript, typed arrays are zero-indexed and writing out of bounds is a no-op. A MATLAB-to-JavaScript compiler must address these mismatches. Another salient and pervasive difference between the two languages is the assignment of arrays to variables: in MATLAB, this operation has value semantics, while in JavaScript is has reference semantics.

In this paper, we present MatJuice — a source-to-source, ahead-of-time compiler back-end for MATLAB— and how it deals efficiently with this last issue. We present an intra-procedural data-flow analysis to track where each array variable may point to and which variables are possibly aliased. We also present the associated copy insertion transformation that uses the points-to information to insert explicit copies when necessary. The resulting JavaScript program respects the MATLAB value semantics and we show that it performs fewer run-time copies than some alternative approaches.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Code generation

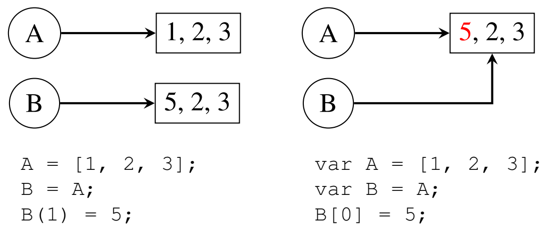
***Keywords*** MATLAB, JavaScript, dataflow analysis, program transformation, dynamic language semantics

*DLS'16*, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4445-6/16/11...$15.00
http://dx.doi.org/10.1145/2989225.2989235

## 1. Introduction

An important semantic difference between MATLAB and JavaScript is how they perform array assignment, pass arrays into functions and return arrays from functions. In MATLAB, those operations are done *by value*, meaning that a full copy of the operand is made before the operation is executed; in JavaScript, those operations are done *by reference*, meaning that memory locations are copied, not the full data.

For example, in MATLAB the assignment statement `B = A` means "make a complete copy of *A* and assign this copy to *B*". The same statement in JavaScript means "take the address of *A* and assign it to *B*". *Figure* 1 illustrates this difference graphically.



```
A = [1, 2, 3];
B = A;
B(1) = 5;
```

```
var A = [1, 2, 3];
var B = A;
B[0] = 5;
```

**(a)** MATLAB: A copied to B **(b)** JavaScript: B points to A

**Figure 1:** Value vs. reference semantics

A MATLAB-to-JavaScript compiler can handle this semantic difference in a few ways. The simplest approach, and one that is obviously correct, is to insert copies at every assignment to array variables in the output program. However, this approach pays a heavy price when arrays need not be copied, i.e., are never mutated. For example, in a matrix multiplication function (`C = mtimes(A, B)`), it would be wasteful to do full copies of *A* and *B*.

A run-time strategy, called copy-on-write, assigns arrays by reference, bumps a counter, and makes a copy only when a write statement is about to modify an aliased variable (when the array's reference count is greater than 1). This approach is used by MathWorks' MATLAB and GNU Octave [5, 14]. This strategy can perform fewer copies than the naive approach, but its main drawbacks are the more complex run-time system it requires and the extra instructions that must be performed at run-time (e.g., reference count updates and checks).

Lameed and Hendren proposed an inter-procedural analysis to remove unnecessary array copies [9]; a naive code generator inserts copies after every array assignment and a subsequent two-phase transformation removes the copies that are proven to be unnecessary. This inter-procedural analysis is implemented in the JIT compiler for MATLAB, McVM.

In this paper, we present MatJuice's new approach to solve this problem: (1) we optimistically assume that no array copies are needed and that assignments are performed by reference as in the target language, (2) we run an intra-procedural analysis to compute the aliasing relationships that exist between the variables of a function, and (3) we run a transformation that inserts the copy statements necessary to obtain the MATLAB value semantics.

We have decided against a copy-on-write approach in MatJuice to keep the run-time system simpler and to avoid introducing extra operations that might interfere with the JavaScript JIT's capacity to optimize a function. We will show that the approach MatJuice adopts performs fewer copies than the naive strategy or a copy-on-write strategy, and is simpler than a whole-program analysis.

This paper presents the following contributions:

1. An intra-procedural points-to analysis that considers the semantics of the target language, rather than the source language;
2. An associated copy-insertion transformation that adds copy statements to bridge the semantics between two dynamic languages;
3. An evaluation of the number of copies and performance of this new approach.

The rest of this paper is organized as follows: in *Section* 2 and *Section* 3 we give a brief overview of MatJuice, and the McLab and Tamer frameworks upon which it builds; in *Section* 4 we show an example MATLAB program and informally describe how we can transform it to retain the proper value semantics in the output JavaScript program; *Section* 5 presents our points-to analysis and shows how information flows between statements in a simple example; *Section* 6 describes how the points-to analysis results are used to insert the necessary copies for local variables and output parameters at the proper program points and also discusses how we use a UseDef analysis to copy input parameters; *Section* 7 investigates how well our transformation performs by benchmarking our approach against alternative strategies; *Section* 8 presents some related work; *Section* 9 concludes and offers ideas for future work.

## 2. Background

In this section we present the McLab and Tamer projects, the two major building blocks of MatJuice.

### 2.1 McLab

McLab is an umbrella project that regroups a large number of compiler-related projects for the MATLAB language: a scanner and parser, a high-level static analysis framework (McSAF), a low-level static analysis framework (Tamer), backends for Fortran and X10, etc. [1–3, 7, 11]

MatJuice is part of McLab, and we use the following components:

- the front-end to scan and parse MATLAB source files and report errors if the source program is syntactically invalid;
- the Tamer analysis framework to implement the points-to analysis described in *Section* 5;
- the visitor patterns to implement the copy insertion transformation, described in *Section* 6;
- the analyses created for other back-ends, such as the shape analysis (obtaining the size and dimensions of a matrix at a given program point), in the code generator;
- the UseDef analysis of the McSAF framework in the implementation of the input parameter copy described in subsection 6.4.

### 2.2 Tamer

The Tamer framework is part of the McLab project; it is such an important building block of MatJuice that it deserves its own section.

The two most important components of Tamer for the implementation of MatJuice are its 3-address code intermediate representation and its analysis framework.

#### 2.2.1 TameIR

The Tamer intermediate representation, called *TameIR*, is a structured 3-address code representation; it is structured because rather than being a linear list of instructions with labels and jumps for control flow, it is organized as a tree. TameIR was created with the explicit goal of making it easier to create static compiler back-ends, which is why we have selected to use it for MatJuice. One way by which this goal is attained is by simplifying a MATLAB program into a limited number of basic constructs. Indeed in MatJuice we need only the following TameIR nodes to compile MATLAB programs into JavaScript:

- Functions definitions
- Assignment statements
    - Copy statement ($x = y$, these are the instructions we need to deal with)
    - Array set ($A(i) = e$)
    - Array get ($x = A(i)$)
    - Call statement ($[a\ b] = f(x)$)
    - Literal assignment ($pi = 3.14$)
- If/else statements
- While loops
- For loops

- `break` and `continue`
- Return statements

Many MATLAB constructs are translated into other basic constructs when the TameIR representation of the program is created; for example *switch/case* statements become a series of *if/else* statements in TameIR and short-circuit operators are appropriately split up. Other statements are helpfully disambiguated. Notably, array accesses and function calls, which have the same syntax in MATLAB, are split into two different node types in TameIR: `TIRArrayGetStmt` and `TIRCallStmt`. This is extremely useful when generating code in a language where the syntax and semantics for those two constructs are different.

### 2.2.2 Analysis Framework

Tamer provides a number of visitors to traverse a TameIR program and to perform data-flow analyses. It is the Tamer framework that is responsible for passing data forward or backward between nodes and for checking that a fixed point has been reached. Tamer also provides simpler visitors (i.e., ones that do not transmit data-flow information between nodes) that are useful for transforming the IR, e.g., adding new nodes.

In MatJuice we use the Tamer analysis and transformation classes for two purposes: (1) to perform our points-to analysis, (2) to insert explicit copies in the body of a function.

In addition, MatJuice reuses some analyses that have already been written for Tamer. One important such analysis is the *shape analysis*: this inter-procedural analysis examines the source code of the input MATLAB program and determines the shape of the variables at each program point. This analysis can tell if a variable is a scalar or an array, and in the case of arrays can tell us the number of dimensions and the size of each dimension. We use this analysis to know which variables are arrays and thus need to be included in our points-to analysis (scalars in JavaScript are assigned by copy and thus need not be considered).

## 3. MatJuice

MatJuice is a new source-to-source, ahead-of-time compiler from MATLAB to JavaScript. *Figure* 2 shows a simplified flow chart of its pipeline architecture; the ellipses represent files on disk, the trapezoid boxes are in-memory data structures, and the rectangle boxes are software modules. The shaded boxes are modules that are part of MatJuice.

**McLab + Tamer:** in this phase, the input program is scanned, parsed and converted to TameIR. Errors are reported if the program is syntactically invalid.

**Points-to analysis:** the TameIR program is given to the points-to analysis which produces a mapping between every TameIR statement of the function and the points-to abstraction that we describe in *Section* 5.

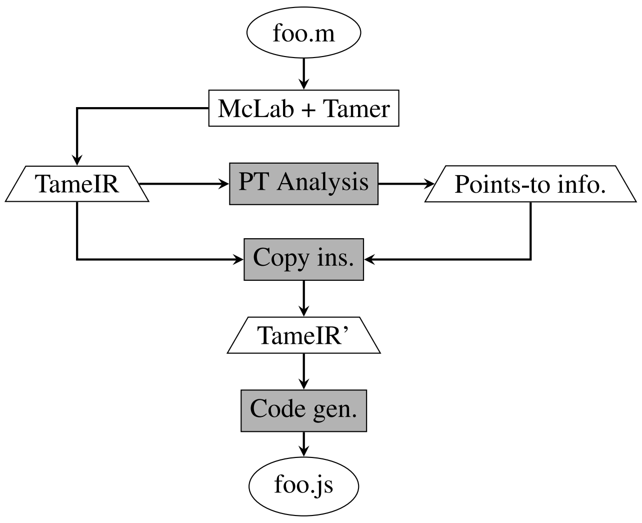


**Figure 2:** MatJuice flow chart

**Copy insertion:** the TameIR program and the result of the points-to analysis are given to this module which produces an updated TameIR program that contains instruction nodes that explicitly copy arrays.

**Code generation:** the copy-inserted program is given to the code generator which translates the different TameIR statements (see subsubsection 2.2.1) into JavaScript and writes the program on disk.

MatJuice represents scalars using JavaScript's *Number* type and all arrays using typed arrays [6]. Typed arrays, like regular JavaScript arrays, have reference semantics, but offer improved performance [4]. By using typed arrays, the code generated by MatJuice is competitive in performance with MathWorks' MATLAB when it is run in Google Chrome or Mozilla Firefox. When no array operations from MATLAB's highly-tuned library (e.g., matrix multiply or LU decomposition) are used in a benchmark, the JavaScript code runs faster than MATLAB.

## 4. Motivating Example

In this section we look at a MATLAB function, optimistically assume that the arrays have the reference semantics of JavaScript, and determine where copies need to be inserted in order to reproduce the value semantics of MATLAB.

The function in Listing 1 accepts two array input parameters, $A$ and $B$, and returns one array, $y$.

### 4.1 Input Parameters

Our first task is to ensure that input parameters are not mutated directly; writes should be performed on copies. We see on line 3 of Listing 1 (`B(1) = -B(1)`) that one of the input parameter, *B*, is modified. Under reference semantics, this statement also mutates the actual argument in the caller of *f*. To prevent this situation from occurring, we must copy *B*

```
 1 function y = f(A, B)
 2     if B(1) < 0
 3         B(1) = -B(1);
 4     end
 5     C = A;
 6     C(1) = 2*C(2);
 7     if A(1)
 8         y = C+B;
 9     else
10         y = A;
11     end
12 end
```

**Listing 1:** Motivating example

before modifying it. A simple and natural place to insert the copy is at the beginning of the function's body. However, we can be more clever: notice that the mutation occurs inside an *if* statement without an *else* block. That means that there exists a path where *B* is not modified and thus need not be copied. We avoid making unnecessary copies (when the condition on line 2 is false) by inserting the copy statement inside the *if* statement.

Since no statement in *f* modifies the input parameter *A*, we don't need to copy it.

### 4.2 Local Variables

Our next task is to make sure that local variables are not aliased when they are modified. On line 5 of Listing 1 (`C = A`), the local variable *C* and the input parameter *A* become aliased. On the next line (`C(1) = 2*C(2)`) the local *C* is modified; because *A* and *C* are aliased, this statement also mutates *A*. To respect the value semantics of MATLAB, we must add a copy of *C* after line 5 where the aliasing relationship was introduced.

### 4.3 Output Parameters

The last task that we need to take care of is making sure that the output parameters are not aliased to externally allocated memory (e.g., input parameters or global variables) and that they aren't aliased to other output parameters.

Let's imagine what would happen in a caller to *f* after the invocation `Z = f(X, Y)`. If the control path in *f* went through line 10, the output parameter *y* would be aliased to the input parameter *A*. Due to our assumption of reference semantics, in the caller, the variables *Z* and *X* would be aliased and writing into either one would mutate the other. To prevent this situation from occurring, we must copy *y* after line 10 (`y = A`). After the copy insertion transformation, all output parameters point to different memory sites.

(Note: if a MATLAB function doesn't have an explicit *return* instruction, the values of the declared output parameters are returned when the control flow reaches the end of the function's body.)

### 4.4 Summary

Listing 2 shows the function *f* after we've inserted the copy statements necessary to obtain the MATLAB semantics (the new copy statements are preceded by a + symbol and highlighted in blue). In addition, we've been able to avoid making some unnecessary copies that a naive code generator would have inserted:

- The input parameter *B* is only copied if the condition on line 2 is true;
- the output parameter *y* is only copied when the condition on line 9 is false.

```
 1   function y = f(A, B)
 2       if B(1) < 0
 3 +         B = copy(B);
 4           B(1) = -B(1);
 5       end
 6       C = A;
 7 +     C = copy(C);
 8       C(1) = 2*C(2);
 9       if A(1)
10           y = C+B;
11       else
12           y = A;
13 +         y = copy(y);
14       end
15   end
```

**Listing 2:** Example function after copy insertion

## 5. Points-to Analysis

In *Section* 4, we relied on our human understanding to know which statements caused variables to become aliased and where to insert the appropriate copy statements. In this section, we present a data-flow analysis that computes information that can be used by an automated process to insert copies in the source program.

At every program point, we will compute a set of $(v, m, s)$ triples; $v$ is a variable name, $m$ is an abstract memory site, and $s$ is a set of aliasing statements that caused $v$ to become aliased with respect to the memory site $m$. Using this abstraction, we can know if two variables are possibly aliased (i.e., if they share at least one memory site) and where to insert copy statements.

### 5.1 Example

Before we go into the technical details and formal presentation of the analysis, let's get an intuitive feel of how the points-to analysis works by looking at the example in Listing 3, and explaining the sets that are computed at every program point. We've annotated the program with comments detailing the data-flow information after each statement. For simplicity, and to fit within the page, we refer to statements

```
 1 function f()        % { }
 2   A = zeros(10);    % { (A, m1, { }) }
 3   B = ones(10);     % { (A, m1, { }) (B, m2, { }) }
 4   if condition      % { (A, m1, { }) (B, m2, { }) }
 5     C = A;          % { (A, m1, {5}) (B, m2, { }) (C, m1, {5}) }
 6   else              % { (A, m1, { }) (B, m2, { }) }
 7     C = B;          % { (A, m1, { }) (B, m2, {7}) (C, m2, {7}) }
 8   end               % { (A, m1, {5}) (B, m2, {7}) (C, m1, {5}) (C, m2, {7}) }
 9   C(1) = 42;        % { (A, m1, {5}) (B, m2, {7}) (C, m1, {5}) (C, m2, {7}) }
10 end                 % { (A, m1, {5}) (B, m2, {7}) (C, m1, {5}) (C, m2, {7}) }
```
**Listing 3:** Points-to information

by their line numbers; in the actual implementation, we use pointers to AST nodes.

On lines 2 and 3 of Listing 3, memory is allocated for the variables *A* and *B* respectively. After the execution of line 3, the variable *A* may point to *m1* and the variable *B* may point to *m2* (our points-to analysis is a *may*-analysis, and so we use the verb "may" when describing a triple, even when a variable definitely points to a given memory site); neither variable has been aliased yet, hence the empty sets in both triples.

On line 5, the assignment statement (`C = A`) causes *C* and *A* to become aliased. A triple is added for *C* in the flow set. In addition, it was at this program point that *A* and *C* started pointing to the same memory sites, and we record this fact by adding statement 5 to the aliasing sets of *A* and *C*. On line 7, the same scenario occurs between *B* and *C*.

On line 8, a merge point, the set computed in the *then* branch is combined with the set computed in the *else* branch. The details of this merge operation are fully explained in subsection 5.2; for now, we can think of this operation as a set union. Once the merge is complete, we have the following information:

- *A* may point to *m1* and it became aliased on line 5;
- *B* may point to *m2* and it became aliased on line 7;
- *C* may point to *m1* or to *m2*; it became aliased with respect to those memory sites on lines 5 and 7.

On line 9, the content of *C* is modified. Looking at the data-flow information at this program point, we see that *C* is possibly aliased to *A* (via *m1*) and to *B* (via *m2*). The sets of aliasing statements associated with *C* indicate where copies must be inserted to break the aliasing relationships: one copy after statement 5, and one copy after statement 7.

Listing 4 shows the data-flow information after the copies have been inserted and the analysis has been executed again. We see that after the execution of the copy statement on line 6, *C* is associated with a fresh memory site, *m3* and no longer may point to *m1*. In addition, statement 5 is removed from the sets of aliasing statements of the triples for *A* and for *C*. After the execution of the copy statement on line 9, an analogous scenario occurs between *B* and *C*.

At the merge point on line 10, we have a data-flow set that says that:

- *A* may point to *m1*;
- *B* may point to *m2*;
- *C* may point to *m3* or to *m4*.

When code for this new function is generated, we know that *C* may point to *m3* or to *m4*, and that it is the only variable that may point to those two locations. Thus the write statement of line 11 cannot affect the content of other variables. No other copy statements are needed, function *f* respects the MATLAB value semantics and is ready for code generation.

## 5.2 Analysis Components

In this section, we formally present our points-to analysis: we describe the abstraction that we use, how flow information is propagated between nodes, how flow information is merged, and what the initial approximations are.

**Approximation** We approximate points-to relationships with a set of triples. The first component of a triple is a variable name $v$; the second component is a memory site $m$; the third component is a set of statements where $v$ became aliased with respect to $m$. The statements where a variable becomes aliased are assignment statements of the form $v = u$; $v$ becomes aliased with $u$ by pointing to $u$'s memory site and $u$ becomes aliased because a new variable ($v$) points to its memory.

A *memory site* is an abstract object that represents one or more zones in memory that have been allocated by a statement for a given variable.

If a variable $v$ may point to more than one memory site, there are multiple triples in the set whose first component is $v$.

**Definition** Let $v$ be an array variable defined at a program point $d$ and pointing to a memory site $m$. We say that the variable $v$ points to $m$ at a given program point $p$ if there

```
  function f()      % { }
    A = zeros(10);  % { (A, m1, { }) }
    B = ones(10);   % { (A, m1, { }) (B, m2, { }) }
    if condition    % { (A, m1, { }) (B, m2, { }) }
      C = A;        % { (A, m1, {5}) (B, m2, { }) (C, m1, {5}) }
+     C = copy(C);  % { (A, m1, { }) (B, m2, { }) (C, m3, { }) }
    else            % { (A, m1, { }) (B, m2, { }) }
      C = B;        % { (A, m1, { }) (B, m2, {7}) (C, m2, {7}) }
+     C = copy(C);  % { (A, m1, { }) (B, m2, { }) (C, m4, { }) }
    end             % { (A, m1, { }) (B, m2, { }) (C, m3, { }) (C, m4, { }) }
    C(1) = 42;      % { (A, m1, { }) (B, m2, { }) (C, m3, { }) (C, m4, { }) }
  end               % { (A, m1, { }) (B, m2, { }) (C, m3, { }) (C, m4, { }) }
```

**Listing 4:** Points-to information after copy insertion

exists at least one path from $d$ to $p$ with no redefinition of $v$.

**Direction** The points-to analysis is a *forward* analysis; information is propagated from statement nodes to their successors.

**Merge operation** A merge node (i.e., a node with multiple predecessors) combines the information from two predecessor nodes $P_1$ and $P_2$ in three steps. We shall call "corresponding triples" a triple $(v, m, s)$ in $P_1$ and a triple $(v', m', s')$ in $P_2$ if $v = v'$ and $m = m'$.

- If $P_1$ and $P_2$ respectively contain the corresponding triples $(v, m, s)$ and $(v, m, s')$, the triple $(v, m, s \cup s')$ is added to the output set;
- Triples in $P_1$ that have no corresponding triple in $P_2$ are added as is to the output set;
- Triples in $P_2$ that have no corresponding triple in $P_1$ are added as is to the output set.

We can express these rules formally using the equation that follows.

$$\begin{aligned} \mathit{out}(S) =& \{(v, m, s \cup s') \mid (v, m, s) \in \mathit{out}(P_1), (v, m, s') \in \mathit{out}(P_2)\} \\ &\cup \{(v, m, s) \mid (v, m, s) \in \mathit{out}(P_1), (v, m, *) \notin \mathit{out}(P_2)\} \\ &\cup \{(v, m, s) \mid (v, m, s) \in \mathit{out}(P_2), (v, m, *) \notin \mathit{out}(P_1)\} \end{aligned}$$

**Starting approximations** The *out* set of the entry node of a function is the set containing triples that map the array input parameters to a common external memory site and an empty set of aliasing statements, $\mathit{out}(\text{ENTRY}) = \{(p, \text{EXTERNAL}, \{\}) \mid p \in \text{input\_params}\}$. Every other statement $S_i$ is approximated by the empty set, $\mathit{out}(S_i) = \{\}$.

**Flow equations** It is common to define the flow equations of an analysis by removing a *kill* set and adding a *gen* set. Although we could use this strategy, we found that it makes the notation long-winded and harder to understand. Therefore, each flow equation is going to define *out(S)* directly in terms of *in(S)*, with the help of some auxiliary definitions. We have found the resulting equations simpler to understand and closer to what a programmer would write in an actual implementation.

**Assign statement** An assignment statement $S$ of the form $A = B$ creates an output set containing the following triples:

- For every memory site $m$ associated with $B$, we create a triple $(A, m, \{S\})$: $A$ may point to any memory site that $B$ may point to and $A$ became aliased at $S$. The previous triples for $A$ are discarded;
- For every triple $(B, m, s)$, we remove all the statements that previously involved $A$ from $s$ and we add the current statement $S$;
- For every triple $(v, m, s)$ where $v$ is neither $A$ nor $B$, we remove all the statements that previously involved $A$ from $s$.

We can express these rules using the following set equation:

$$\begin{aligned} &\mathit{stmtsA} = \bigcup \{s \mid (A, *, s) \in \mathit{in}(S)\} \\ &\mathit{memsitesB} = \{m \mid (B, m, *) \in \mathit{in}(S)\} \\ &\mathit{out}(S) = \{(A, m, \{S\}) \mid m \in \mathit{memsitesB})\} \\ &\quad \cup \{(B, m, (s - \mathit{stmtsA}) \cup \{S\}) \mid (B, m, s) \in \mathit{in}(S)\} \\ &\quad \cup \{(v, m, s - \mathit{stmtsA}) \mid (v, m, s) \in \mathit{in}(S), \\ &\qquad v \neq A, v \neq B\} \end{aligned}$$

**Assign literal** An assign statement $S$ of the form $A = x$ where $x$ is a scalar literal creates the following output set:

- All the triples $(v, m, s)$ where $v$ is not $A$ are included in the output set and we remove all the statements that previously involved $A$ from $s$.
- No triples for $A$ are included.

$$stmtsA = \bigcup\{s \mid (A, *, s) \in in(S)\}$$
$$out(S) = \{(v, m, s - stmtsA) \mid (v, m, s) \in in(S), v \neq A\}$$

**Function calls** A function call statement $S$ of the form $[r_1\ r_2 \ldots r_n] = f(a_1, a_2, \ldots, a_k)$ is similar to the assignment statement except that instead of having a single variable on the left-hand side, we possibly have multiple ones. As we noted in subsection 4.3, we impose the invariant that all output parameters point to distinct memory sites.

- A triple $(r_i, m, \{\})$ is added to $out(S)$ for every variable $r_i$ on the left-hand side. The memory site $m$ is a function of the statement and the variable: given the same statement and variable, we should obtain the same memory site, and that memory site must be unique (i.e., there is no other statement-variable pair that yields the same memory site).
- The triples $(v, m, s)$, where $v$ is not one of the left-hand side variables, from $in(S)$ are included and we remove all statements involving any of the $r_i$ from $s$.

$$newSite(stmt, var) = \text{unique memsite for (stmt, var)}$$
$$aliasStmts = \bigcup\left(\bigcup_{i=1}^{i=n}\{s \mid (r_i, *, s) \in in(S)\}\right)$$
$$out(S) = \{(r_i, newSite(S, r_i), \{\}) \mid i \in 1..n\}$$
$$\cup\, \{(v, m, s - aliasStmts) \mid (v, m, s) \in in(S), v \notin \{r_1, \ldots, r_n\}\}$$

The function $newSite$ is necessary in order for a fixed point to be reached, i.e., that the sets computed in iteration $k$ are all equal to the sets computed in iteration $k-1$. If an entirely new memory site was given at every iteration, the fixed point procedure would diverge.

**Other statements** A statement $S$ of any other type lets the information flow through unchanged. $out(S) = in(S)$

## 6. Copy Insertion Transformation

Now that we have points-to sets at each program point of a function, let's use use that information to insert the copy statements necessary to obtain the MATLAB value semantics. After the function is transformed, the following three properties will hold:

**Property 1** If there exists a statement $S$ that modifies an input parameter $p$, then all paths leading to $S$ contain a statement that copies $p$;

**Property 2** If there exists a statement $S$ that writes into a local variable $x$, then $x$ is not aliased at $S$;

**Property 3** If there exists a statement $S$ that returns the output parameters $o_1$, $o_2$, ..., $o_k$, then none of the output parameters are aliased to one another or possibly point to externally allocated memory at $S$.

Properties 1 and 3 are particularly important in the context of an intra-procedural analysis. Suppose a MATLAB programs contains the following statement: `[a b] = f(x)`. After the function call, a MATLAB programmer has a few expectations, notably that the content of *x* hasn't changed and that *a*, *b*, and *x* are independent and mutating one won't affect the others.

Property 1 ensures that the input parameter *x* will be the same before and after the call to *f*; if *x* is modified in the body of the function, we copy it. An inter-procedural analysis could detect that it's unnecessary to copy *x* if it's dead, i.e., never used after the call to *f*. However, in MatJuice's intra-procedural analysis we cannot tell what happens outside the function and so we must conservatively assume that *x* is live.

Similarly, property 3 ensures that *a* and *b* aren't aliased to one another, and that neither is aliased to *x* or to a global variable. This is accomplished by having the transformation make sure that all output parameters point to memory allocated inside *f* (i.e., memory that the intra-procedural analysis has knowledge of) and that they aren't aliased amongst themselves. Again, in an inter-procedural analysis with the knowledge of how $a$ and $b$ are used, the transformation could decide that it's fine for them to be possibly aliased and forgo a copy.

### 6.1 General Process

The copy insertion transformation is a fixed-point algorithm and is invoked at the beginning of MatJuice's code generation phase. Before a MATLAB function is translated to JavaScript, it goes through the following loop:

1. Add copies for input parameters which may be modified;[1]
2. Apply the points-to analysis to the function;
3. Add copy statements for locals and output parameters;
4. If step 3 inserted at least one copy statement, goto 2.

At each iteration, after copies for one variable have been added, we terminate the transformation process and run the points-to analysis again. To understand why, let's look at the simple example in Listing 5.

If the copy insertion tried to add copies for all variables at once, it would find that statement 4 writes into *A* which is possibly aliased to *B* and it would add a copy after line 3. This copy would break the aliasing relationship that exists between *A* and *B* as they now point to distinct memory locations. If the transformation were allowed to continue, by looking at the now-stale information of statement 5, it would find that *A* and *B* are possibly aliased and add a copy for *B* after line

[1] subsection 6.4 explains why input parameters are not part of the analysis-transformation loop.

```
function f()     % {}
  A = zeros(2); % {(A,m1,{ })}
  B = A;        % {(A,m1,{3}) (B,m1,{3})}
  A(1,1) = 10;  % {(A,m1,{3}) (B,m1,{3})}
  B(2,1) = 20;  % {(A,m1,{3}) (B,m1,{3})}
end
```

**Listing 5:** Copying one variable at a time

3, thus inserting an unnecessary operation. Terminating the transformation and re-executing the points-to analysis ensures that the most up-to-date information is always available.

The analysis-transformation process is guaranteed to terminate: in the worst case, a copy statement is added after every assignment statement. At that point, no variables will be possibly aliased and no more copies will be inserted.

## 6.2 Copying Local Variables

The copy insertion transformation visits all the statements of a function; when an array write statement of the form `A(i) = e` is found, we inspect the points-to information at that program point. If we find that *A* is possibly aliased, meaning that it shares at least one memory site with another variable, we insert a copy for *A* after every statement contained in the set of aliasing statements for the offending memory sites.

This transformation inserts copies only when it finds a write statement; if some variables are possibly aliased but no write statements are performed on any of them, no copies will be inserted. This is in line with our goal of not inserting unnecessary copies. This transformation also ensures that property 2 holds; if the variable *A* was not aliased, then the assignment doesn't affect other variables; if *A* was aliased, we've now inserted copy statements between the aliasing points and the assignment, thus making sure that *A* is no longer aliased.

## 6.3 Copying Output Parameters

When the copy insertion transformation finds a return statement, it inspects the points-to information of the output parameters. If an output parameter *p* may point to externally allocated memory or is possibly aliased to another output parameter, copies of *p* are inserted in the function at the program points indicated by the aliasing statements set for the offending memory sites.

In MATLAB, if the control flow falls off the end of the function's body, the value of the output parameters is returned to the caller. To make this case easier to deal with, we insert a return statement at the end of the function before performing the points-to analysis.

This transformation ensures property 3, by making sure that all the output parameters point to locally-allocated memory and that no two output parameters point to the same memory.

## 6.4 Copying Input Parameters

In the previous sub-sections, we discussed how to use the points-to analysis results to find the output parameters and the local variables that are possibly aliased and how the copy insertion transformation uses that information to add the necessary copy statements. We now address the issue of input parameters.

After the points-to analysis, one of the facts that the data-flow information allows us to determine is which local variables and output parameters are possibly aliased. In the case of input parameters, we *know* that they are aliased: the array input parameters always refer to memory that was allocated in another function of the program, i.e., the actual arguments passed in the function call. The question is not to know if input parameters are aliased, but rather (1) if they need to be copied, (2) where those copies should be inserted.

To answer the first question, which input parameters need to be copied, we can re-use McLab's UseDef analysis. An array write statement of the form `A(i) = e` modifies an input parameter *p* if *p* is contained in the UseDef set of *A*.

*Algorithm* 1 describes how to compute the input parameters that need to be copied. If the set of statements associated with an input parameter *p* is empty, the input parameter is not modified and need not be copied (the purpose of this set of statements will be explained shortly). If the set of statements is not empty, a copy statement for *p* needs to be added.

**Algorithm 1:** Analysis for input parameters

```
paramsToCopy ← emptyMap
for all $p \in$ input parameters do
  stmts ← {}
  for all $s \in$ function statements do
    if $s$ has the form A(i)=e and $p \in$ UseDef(A) then
      stmts ← stmts $\cup \{s\}$
    end if
  end for
  if stmts $\neq$ {} then
    paramsToCopy.put($p$, stmts)
  end if
end for
```

The second question, where should an input parameter *p* be copied, is answered by using the sets computed in *Algorithm* 1. A copy statement is added at the beginning of the inner-most block that is a common ancestor to all the statements in the set associated with *p*. In addition, this block must not be inside a loop. By pushing the copy as deep within the function as possible, we can avoid making unnecessary copies when the dynamic control flow doesn't reach any of the write statements for a given input parameter (as we saw in Listing 2). Putting the copies outside of loops is necessary to respect the semantic of MATLAB which performs only one copy per input parameter.

This transformation ensures that property 1 is respected; if the input parameter $p$ is modified at statement $S$, there will definitely be a copy statement between the function's entry point and $S$.

## 7. Evaluation

In this section we evaluate MatJuice's copy insertion using three metrics:

**Metric 1:** the number of copies performed at run-time;

**Metric 2:** the impact of those copies on execution time;

**Metric 3:** the impact of the analysis and transformation on compilation times.

We first investigate Metrics 1 and 2 by comparing MatJuice's copy insertion against a modified implementation of MatJuice that copies everything (the naive strategy).

We then compare MatJuice's copy insertion against a copy-on-write strategy; since MathWorks' MATLAB is closed-source, we do the comparison against GNU Octave, a free software implementation compatible with MATLAB. Like MathWorks' MATLAB, GNU Octave uses a copy-on-write system to copy arrays. Because it is difficult to meaningfully measure the performance impact of a single implementation detail in two different systems, we only consider Metric 1 in this part of the evaluation.

We use the set of benchmarks described in *Table* 1 for our evaluation. We chose these benchmarks because they cover a wide variety of problems from numerical computing and because most of them make heavy use of two programming constructs that are important in MATLAB: arrays and for loops. Benchmarking against those features should give us a fairly good idea of how MatJuice-generated code will perform in other programs. In addition, some of these benchmarks have been used in the past by Lameed and Hendren [8] in the evaluation of their inter-procedural copy removal transformation.

### 7.1 Naive Copy vs. Copy Insertion

Let us first examine how MatJuice's copy insertion improves over the naive strategy. *Table* 2 shows the figures we've obtained. The "Naive" columns report the figures when copies are always performed and the "CI" columns report the figures when points-to analysis and copy insertion are performed. The "copies" column shows how many calls to *mj_clone*, the method that performs a full copy of an array, were performed. Each benchmark was executed multiple times, specified in the "scale" column, in order for the timings to be high enough to be comparable.

The first thing we notice from those figures is that in all benchmarks, the number of copies lower when copy insertion is enabled. For some benchmarks (e.g., *bubble*, *clos*, *fft*, and *matmul*) the number of copies goes from a handful to one or zero; in those cases, the naive implementation copies the array input parameters while the copy insertion implementation the analysis is able to determine that such a copy is unnecessary. The low number of copies is due to the nature of those benchmarks: *clos* and *prime* compute a scalar value, while *bubble* and *matmul* perform a single iteration (the scale factor in this case represents the size of the input, e.g., multiplying two $400 \times 400$ matrices).

For other benchmarks—such as *capr*, *crni*, and *nb1d*—the difference is more drastic: copy insertion reduces the number of copies by tens of thousands in the more extreme cases. For *capr* and *crni*, the reduction in copies comes solely from the input parameter analysis being able to determine that some of the input parameters are not modified during the execution of the function and thus need not be copied. For *fdtd*, the conversion from MATLAB to TameIR introduced extra assignment in temporaries and MatJuice's points-to analysis can determine that these variables are read-only (i.e., no write is ever performed on these arrays) and need not be copied. For *nb1d*, both the input parameter analysis and the points-to analysis determine that the copies are unnecessary.

Some benchmarks, such as *babai* and *lgdr* show a large reduction in the number of copies: this is due to the high scale factor (i.e., the benchmark is executed many times over). In actual fact, a single execution of *babai* performs two copies and one execution of *lgdr* performs one.

The effects of the copy insertion transformation on execution time are apparent: for *fdtd* and *capr*, the smarter translation gives a speedup of 11%, and *nb1d* obtains a 17% speedup. In the case of *crni*, the speedup is a more modest 1%. We also observe a correlation between the "copies per second" column and the "speedup" column; the most significant speedups came from benchmarks that were copy-intensive, such as *capr*, *makechange*, and *nb1d*.

### 7.2 Comparison with Octave

In this section, we look at how MatJuice's compile-time approach compares to a run-time approach, copy-on-write. Copy-on-write is a system where an array is copied just before it is modified and only if it is aliased. We have used GNU Octave 4.0 as our comparison point because MathWorks' MATLAB is proprietary and closed source and thus it was not possible to get the source code and instrument it.

In *Table* 3, we show the number of copies performed at run-time by GNU Octave and MatJuice with and without copy insertion. For Octave, we have also included a column to show the number of reference count checks made by the `make_unique` function in the file `liboctave/array/-Array.h`.

A number of our benchmarks were also used in Lameed and Hendren [8]. In that paper, the authors computed, using AspectMatlab, a lower bound on the number of copies necessary to obtain the value semantics of MATLAB. For those shared benchmarks, we've included the lower bound that they computed.

We notice in the table that the number of copies performed by the copy-on-write and the copy-insertion systems are

| Benchmark | Source | Description |
|---|---|---|
| babai | MATLAB file exchange | Compute the Babai estimation for an integer least square problem |
| bubble | McLab | Bubble sort, a $O(n^2)$ sorting algorithm |
| capr | Chalmers University | Compute the capacitance of a transmission line using finite difference and Gauss-Seidel iteration |
| clos | Otter project | Compute the transitive closure of a directed graph |
| collatz | McLab | Test the Collatz conjecture up to a given integer |
| crni | Falcon project | Compute the Crank-Nicholson solution to the one-dimensional heat equation |
| dich | Falcon project | Compute the Dirichlet solution to Laplace's equation |
| fdtd | EEK 170 | Apply the Finite Difference Time Domain (FDTD) technique on a hexahedral cavity with conducting walls. |
| fft | Press et. al | Compute the discrete Fourier transform |
| fiff | Falcon project | Compute the finite-difference solution to a given wave equation |
| lgdr | Unknown | Compute the normalized, orthogonormal Legendre polynomials $P_n(x)$ for all degrees up to and including $n$ and their first and second derivatives |
| makechange | McLab | Compute the ways to make change for a using dynamic programming |
| matmul | McLab | Naive $O(n^3)$ matrix multiplication |
| mcpi | McLab | Calculate $\pi$ by the Monte Carlo method |
| nb1d | Otter project | Simulate the 1-dimensional n-body problem |
| numprime | Burkardt and Cliff | Count the number of primes up to a given integer |

**Table 1:** List of benchmarks

| | MatJuice (Naive) | | | MatJuice (CI) | | | |
|---|---|---|---|---|---|---|---|
| Benchmark (scale) | Time (s) | Copies | Copies/s | Time (s) | Copies | Copies/s | Speedup |
| babai (2000) | $0.58 \pm 0.00$ | 4000 | 6896.55 | $0.56 \pm 0.01$ | 0 | 0.00 | 1.04 |
| bubble (10,000) | $3.49 \pm 0.03$ | 2 | 0.57 | $3.47 \pm 0.01$ | 1 | 0.29 | 1.01 |
| capr (5) | $8.41 \pm 0.02$ | 150000 | 17835.91 | $7.57 \pm 0.01$ | 50000 | 6605.02 | 1.11 |
| clos (1) | $19.10 \pm 0.02$ | 1 | 0.05 | $19.37 \pm 0.03$ | 0 | 0.00 | 0.99 |
| collatz (1,000,000) | $2.46 \pm 0.05$ | 0 | 0.00 | $2.48 \pm 0.04$ | 0 | 0.00 | 0.99 |
| crni (5) | $16.15 \pm 0.02$ | 45980 | 2847.06 | $15.93 \pm 0.04$ | 22990 | 1443.19 | 1.01 |
| dich (5) | $4.96 \pm 0.01$ | 0 | 0.00 | $4.97 \pm 0.02$ | 0 | 0.00 | 1.00 |
| fdtd (1) | $19.14 \pm 0.59$ | 600 | 31.35 | $17.32 \pm 0.56$ | 0 | 0.00 | 1.11 |
| fft (9) | $1.17 \pm 0.01$ | 2 | 1.71 | $1.16 \pm 0.00$ | 1 | 0.86 | 1.01 |
| fiff (5) | $5.53 \pm 0.03$ | 0 | 0.00 | $5.52 \pm 0.02$ | 0 | 0.00 | 1.00 |
| lgdr (1000) | $1.02 \pm 0.01$ | 3000 | 2941.18 | $1.02 \pm 0.01$ | 0 | 0.00 | 1.00 |
| makechange (2000) | $1.00 \pm 0.04$ | 2001 | 2001.00 | $0.89 \pm 0.00$ | 0 | 0.00 | 1.12 |
| matmul (400) | $1.67 \pm 0.01$ | 2 | 1.20 | $1.62 \pm 0.01$ | 0 | 0.00 | 1.03 |
| mcpi (1,000,000) | $1.67 \pm 0.07$ | 0 | 0.00 | $1.67 \pm 0.06$ | 0 | 0.00 | 1.00 |
| nb1d (5) | $6.34 \pm 0.05$ | 198202 | 31262.15 | $5.40 \pm 0.04$ | 0 | 0.00 | 1.17 |
| numprime (5,000,000) | $4.28 \pm 0.12$ | 0 | 0.00 | $4.41 \pm 0.09$ | 0 | 0.00 | 0.97 |

**Table 2:** MatJuice without copy insertion (Naive) vs. MatJuice with copy insertion (CI)

| | Octave | | MatJuice (Naive) | MatJuice (CI) | |
|---|---|---|---|---|---|
| Benchmark (scale) | Copies | Ref. checks | Copies | Copies | Lower Bound † |
| babai (1) | 2 | 1781 | 2 | 0 | N/A |
| bubble (1000) | 1 | 11,828,685 | 1 | 2 | N/A |
| capr (1) | 10,002 | 671,159,808 | 30,000 | 10,000 | 10,000 |
| clos (1) | 0 | 1,855,124 | 1 | 0 | 0 |
| collatz (1000) | 0 | 813,759 | 0 | 0 | N/A |
| crni (1) | 4601 | 460,060,064 | 9196 | 4598 | 4598 |
| dich (1) | 2 | 471,926,535 | 0 | 0 | 0 |
| fdtd (1) | 0 | 153,604 | 600 | 0 | 0 |
| fft (1) | 1 | 836,619 | 2 | 1 | 1 |
| fiff (1) | 0 | 587,737,668 | 0 | 0 | N/A |
| lgdr (1) | 0 | 1140 | 3 | 0 | N/A |
| makechange (1000) | 0 | 418 | 2 | 0 | N/A |
| matmul (256) | 0 | 251,791,779 | 2 | 0 | N/A |
| mcpi (1) | 0 | 500,143 | 2 | 0 | N/A |
| nb1d (1) | 8 | 3,973,171 | 4601 | 0 | 0 |
| numprime (10000) | 0 | 32,175 | 0 | 0 | N/A |

**Table 3:** Number of array copies performed at run-time — † Source: Lameed and Hendren [8]

extremely similar: in 11 of the benchmarks, they are identical and in 5 benchmarks copy insertion actually performs fewer copies.

In the introduction of this paper, we mentioned that we rejected using a copy-on-write approach, partly in order to avoid performing extra instructions at run-time. The "Ref. checks" column confirms that this was a good idea: in some benchmarks, Octave performs hundreds of millions of branching instructions. It is a testament to the efficacy of our copy insertion approach that we are able to generate code that performs fewer copies without the overhead of reference counting.

In addition, for the benchmarks where we have a lower bound on the necessary number of copies, we see that the number of copies performed by MatJuice with copy insertion is equal. This is a very strong indication of the effectiveness of MatJuice's technique.

### 7.3 Time Benchmarks

We ran the benchmarks from *Table* 2 without instrumentation using Node.js (v4.4.0), MathWorks' MATLAB (2015b), and GNU Octave (4.0) to see the difference in performance between these three implementations. The complete data is shown in *Table* 4.

The bout between MATLAB and JavaScript was pretty even; JavaScript was faster on 10 of the 16 benchmarks, but on average MATLAB was faster by about 20%. The best MATLAB program (*fdtd*) was 180 times faster than JavaScript, and the slowest one (*numprime*) was 11 times slower than JavaScript.

JavaScript is faster than GNU Octave on 12 of the 16 benchmarks. On average, Octave was 77 times slower than JavaScript. In 10 of the benchmarks, Octave was 100 times slower than JavaScript, and in 8 of those benchmarks, it was in fact more than 1000 times slower; in two benchmarks (*fiff* and *matmul*), the slowdown factor exceeds 5000x. JavaScript is slower than Octave in the benchmarks where a loop contains a matrix multiply operation: the naive JavaScript implementation of MatJuice cannot compete with the highly-tuned numeric procedures used by Octave.

### 7.4 Compile Time

The overhead of the points-to analysis and the copy insertion transformation is detailed in *Table* 5.

The first two columns show the number of functions in the benchmark and the number of iterations that were required to reach a fixed point in the transformation. We can see that in all but two functions (*capr* and *nb1d*), the number of iterations is equal to the number of functions, meaning that a single pass was necessary to properly transform the code.

We also computed the total time (in seconds) it took to compile a program, and the time it took for the analysis and transformation loop to complete. The percentage of time taken by the analysis/transformation stage takes 5.7% of the total compilation time in the best case (*collatz* and *prime*) and 30.5% in the worst case (*fdtd*).

| | JavaScript | MATLAB | | Octave | |
|---|---|---|---|---|---|
| Benchmark | Time (s) | Time (s) | Ratio | Time (s) | Ratio |
| babai | 0.87 | 0.38 | 0.44 | 0.09 | 0.10 |
| bubble | 0.45 | 1.42 | 3.14 | 1078.55 | 2393.06 |
| capr | 0.78 | 2.23 | 2.84 | 2112.64 | 2691.60 |
| crni | 6.47 | 2.16 | 0.33 | 1800.89 | 278.16 |
| clos | 18.80 | 0.29 | 0.02 | 0.77 | 0.04 |
| collatz | 2.37 | 6.36 | 2.68 | 1421.78 | 599.37 |
| dich | 0.64 | 1.38 | 2.16 | 1667.36 | 2609.33 |
| fdtd | 16.88 | 0.09 | 0.01 | 0.17 | 0.01 |
| fft | 0.09 | 0.35 | 4.06 | 266.09 | 3112.19 |
| fiff | 0.35 | 1.95 | 5.52 | 1831.51 | 5173.76 |
| lgdr | 0.98 | 0.18 | 0.18 | 0.83 | 0.84 |
| makechange | 0.11 | 0.34 | 3.02 | 298.21 | 2684.13 |
| matmul | 0.11 | 0.72 | 6.27 | 681.32 | 5960.83 |
| mcpi | 1.62 | 2.41 | 1.49 | 23.83 | 14.74 |
| nb1d | 5.30 | 1.07 | 0.20 | 17.42 | 3.29 |
| numprime | 3.92 | 43.88 | 11.20 | 5435.18 | 1387.48 |

**Table 4:** Time comparison between JavaScript, MATLAB, and Octave; ratios are relative to JavaScript

| Benchmark | Func. | Iter. | PT/CI (s) | Total (s) | Pct. |
|---|---|---|---|---|---|
| babai | 2 | 2 | 0.10 | 1.20 | 8.4% |
| bubble | 2 | 2 | 0.09 | 1.19 | 7.9% |
| capr | 5 | 6 | 0.22 | 1.83 | 12.1% |
| clos | 2 | 2 | 0.10 | 1.24 | 8.0% |
| collatz | 2 | 2 | 0.07 | 1.19 | 5.7% |
| crni | 3 | 3 | 0.19 | 1.63 | 11.8% |
| dich | 2 | 2 | 0.35 | 2.32 | 15.2% |
| fdtd | 2 | 2 | 0.76 | 2.48 | 30.5% |
| fft | 2 | 2 | 0.42 | 2.11 | 19.6% |
| fiff | 2 | 2 | 0.24 | 1.61 | 15.0% |
| lgdr | 4 | 4 | 0.18 | 1.58 | 11.3% |
| makechange | 2 | 2 | 0.11 | 1.26 | 8.4% |
| matmul | 2 | 2 | 0.07 | 1.15 | 6.4% |
| mcpi | 2 | 2 | 0.08 | 1.19 | 7.0% |
| nb1d | 3 | 6 | 0.26 | 1.78 | 14.8% |
| prime | 2 | 2 | 0.07 | 1.14 | 5.7% |

**Table 5:** Overhead of analysis and transformation

## 8. Related work

Lameed and Hendren have described a two-phase analysis that was implemented in McVM, a JIT compiler for MATLAB [9]. The first phase of McVM is similar to MatJuice's input parameter copy insertion: it find which input parameter definitions reach array update statements and the parameters that are so modified are copied at the beginning of the function's body. The main improvement of MatJuice is to find the deepest block where a copy can be inserted without affecting the MATLAB semantics, thus avoiding copies on some execution paths.

GNU Octave uses a copy-on-write system to handle copies [5]; arrays have a `count` field that indicates how many variables point to that data object. When an array modification occurs, if the reference count is greater than one, the array is copied before the write occurs. Though the MathWorks'

MATLAB source code cannot be inspected to confirm, articles from MathWorks and other sources on the Internet indicate that MathWorks' MATLAB uses the same strategy [10, 14]; in addition some simple experiments strongly suggest that arrays are not copied at assignment statements, but at array update statements.

The R programming language [12] — a dynamic, array-oriented programming language used extensively in the world of statistics — also uses a copy-on-write strategy [13]. One difference from MATLAB that is noted in the R documentation is that rather than maintaining a reference count, an array in R has a field (called `named`) that indicates whether the array is aliased or not.

## 9. Conclusion

In this paper, we showed a new way to bridge the value semantics of MATLAB arrays with the reference semantics of JavaScript arrays. We introduced an intra-procedural data-flow analysis that determines at each program point where the different variables may point to, and a set of the statements where a variable became aliased with respect to a memory site. This analysis uses the unusual trick of considering the MATLAB program as if it had the semantics of JavaScript. We presented the associated copy insertion transformation that accepts a TameIR function and points-to information to insert the necessary copy instructions for the translated MATLAB program to have the the proper semantics.

The figures obtained in *Section* 7 show that MatJuice's points-to analysis and copy insertion transformation can dramatically reduce the number of copies that need to be performed at run-time, and in fact we've seen that MatJuice is able to match the number of copies of a run-time system and matches the lower-bound of copies necessary for a number of our benchmarks. These results tell quite clearly that an intra-procedural approach that adds no extra run-time machinery is definitely suitable for the kind of numerical problems typical of MATLAB. Also, by not needing a reference counting system in the output program, we save CPU cycles and introduce less code that may prevent a JIT compiler from performing aggressive optimizations.

The copy insertion transformation could be improved by using liveness information; if at a statement `A(i) = e` the variable *A* is possibly aliased, but the variables it is aliased to are not live, we could avoid introducing an unnecessary copy statement.

## References

[1] A. Casey et al. McLab: an extensible compiler toolkit for MATLAB and related languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 114–117, New York, NY, USA, 2010. ACM. URL `http://doi.acm.org/10.1145/1822327.1822343`.

[2] J. Doherty. McSaf: An Extensible Static Analysis Framework For The MATLAB Language. Master's thesis, August 2011.

[3] A. W. Dubrau and L. J. Hendren. Taming MATLAB. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 503–522, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384653. URL `http://doi.acm.org/10.1145/2384616.2384653`.

[4] Faiz Khan et al. Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies. URL `http://doi.acm.org/10.1145/2775052.2661090`.

[5] GNU. Octave: Miscellaneous Techniques. `https://www.gnu.org/software/octave/doc/interpreter/Miscellaneous-Techniques.html`.

[6] T. K. Group. Typed Array Specification v1.0. `https://www.khronos.org/registry/typedarray/specs/1.0`.

[7] V. Kumar. Mix10: Compiling matlab to x10 for high performance. Master's thesis, April 2014.

[8] N. Lameed and L. Hendren. Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. In *Proceedings of the 20th International Conference on Compiler Construction*. Springer Berlin / Heidelberg, 2011.

[9] N. Lameed and L. Hendren. Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. In *ETAPS 2011*. Springer Berlin / Heidelberg, Mar 2011.

[10] J. Lecoq. Matlabtips: Copy on Write. `http://www.matlabtips.com/copy-on-write/`.

[11] X. Li. Mc2for: A tool for automatically translating matlab to fortran 95. pages 234–243. IEEE, 2014. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6747175`.

[12] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, . URL `https://www.R-project.org`.

[13] R Core Team. R Internals: Rest of header. `https://cran.r-project.org/doc/manuals/r-patched/R-ints.html#Rest-of-header`, .

[14] L. Shure. MATLAB: Memory Management for Functions and Variables. `http://blogs.mathworks.com/loren/2006/05/10/memory-management-for-functions-and-variables/`.