# Safety at Speed

## In-place array algorithms from pure functional programs by safely re-using storage

Markus Aronsson
Koen Claessen
Mary Sheeran
Nicholas Smallbone
{mararon,koen,ms,nicsma}@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

## ABSTRACT

We present a purely functional array programming language that offers safe, purely functional and crash-free in-place array transformations. The language supports high-level abstractions for pure and efficient array computations that fully support equational reasoning. We show how to execute selected parts of these computations *safely* in-place, with the compiler guaranteeing that in-place execution does not change the computation's result. Correctness is ensured by using an off-the-shelf-theorem prover to discharge safety conditions. Our main contribution is the idea of virtual copies for expressing re-use of arrays, and techniques for verifying their safety, which allow a pure language to include in-place transformations without weakening its transparency or reasoning power.

**ACM Reference Format:**
Markus Aronsson, Koen Claessen, Mary Sheeran, and Nicholas Smallbone. 2019. Safety at Speed: In-place array algorithms from pure functional programs by safely re-using storage. In *FHPNC '19: ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, August 18, 2019, Berlin.* ACM, New York, NY, USA, 13 pages. https://doi.org/???

## 1 INTRODUCTION

Functional programmers define their array programs as pure functions operating on immutable arrays. By doing so, they can program by composing high-level combinators such as *map*—and compilers can transform this high-level pure code into efficient, low-level, mutating code.

When writing code in this style, the programmer loses control of memory allocation and evaluation order, which is left to the compiler. This makes it impossible to define *in-place array algorithms.* Many pure array functions can be safely computed in place, but to do so correctly the programmer must closely control the order in which reads and writes are executed. Since this is at odds with the normal style of functional array programming, in-place algorithms are written in a low-level imperative style. The result is that writing in-place algorithms is a difficult, error-prone job: it is easy to overwrite the input data too early, but the programmer gets no help avoiding such mistakes.

In this paper we present a high-level, functional approach to defining in-place array algorithms. The programmer first writes their algorithm in a normal, functional style, as a pure function. They then specify which computations should be done in place, and any special evaluation order that is needed. The compiler checks that performing the computations in place does not change the behaviour of the program. We have implemented our approach in the co-Feldspar [2] language, but the ideas can be used in any array language that compiles to imperative code. We validate our approach by using it to safely define an in-place FFT, an algorithm that requires an unusual evaluation order.

### 1.1 A First Example

We start by showing how the programmer can use our system to create an in-place array algorithm. We will use the following running example, a co-Feldspar program which combines two arrays:

```
example :: IArr Int32 → IArr Int32 → M (IArr Int32)
example arr1 arr2 =
  manifestFresh (zipWith (∗) (map succ arr1) (reverse arr2))
```

The *manifestFresh* combinator evaluates its argument and stores the result into a freshly-allocated array. In co-Feldspar, memory allocation is made explicit by combinators like *manifestFresh*, and all other array operations are fused—so this program compiles to the following efficient imperative code:

```
result := new int32[n];    -- assume len(arr1) = len(arr2) = n
for i := 0 to n − 1 do
  result[i] := (arr1[i] + 1) ∗ arr2[n − i − 1]
```

Looking at the imperative code, this function could just as well run in place, writing the result back to *arr1* instead of a new array. To specify this, we replace the use of *manifestFresh* with *manifestReuse arr1*:

```
example :: IArr Int32 → IArr Int32 → M (IArr Int32)
example arr1 arr2 =
  manifestReuse arr1
    (zipWith (∗) (map succ arr1) (reverse arr2))
```

The *manifestReuse* combinator evaluates its second argument and stores the result into the array given in the first argument, re-using

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*FHPNC 2019, August 18, 2019, Berlin*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN ???…$???
https://doi.org/???

the first array instead of allocating fresh storage. But it also makes sure that this storage re-use is *safe*, by checking that the program behaves *as if* a fresh array had been allocated. If the compiler accepts a use of *manifestReuse*, it is guaranteed to have the same semantics as *manifestFresh*. The compiler accepts the safety of the program above (provided that the caller of *example* doesn't use *arr1* afterwards), and then emits the following code:

```
for i := 0 to n − 1 do
  arr1[i] := (arr1[i] + 1) ∗ arr2[n − i − 1]
```

If we instead wrote *manifestReuse arr2*, then the compiler would reject the program, because the resulting code would not calculate the same function as the original program:

```
for i := 0 to n − 1 do
  arr2[i] := (arr1[i] + 1) ∗ arr2[n − i − 1]
```

In order to safely reuse *arr2*, the generated loop must write $arr2[i]$ and $arr2[n-i-1]$ simultaneously, much like the algorithm for in-place array reversal. We specify this write order with the combinator *pairwise* $(\lambda i.\,(i, n-i-1))$:

```
example :: IArr Int32 → IArr Int32 → M (IArr Int32)
example arr1 arr2 =
  manifestReuse arr2
    (pairwise (λi. (i, n − i − 1))
      (zipWith (∗) (map succ arr1) (reverse arr2)))
```

The compiler verifies that this code can safely be executed in place, and then emits the following code, in which indexes $i$ and $n-i-1$ of the result array are first computed and then simultaneously written:

```
for i := 0 to (n − 1) / 2 do begin
  x := (arr1[n − i − 1] + 1) ∗ arr2[i];
  y := (arr1[i] + 1) ∗ arr2[n − i − 1];
  arr2[i] := y;
  arr2[n − i − 1] := x
end
```

Here is the point: *even though this function is executed in place, it is still pure.* Since the compiler checks that executing our function in place does not change its semantics, we can mentally replace all occurrences of *manifestReuse* with *manifestFresh* when reasoning about the program. We can freely use equational reasoning to understand or develop our program, without worrying about side effects. The semantics of the in-place *example* function above is precisely that of *zipWith* (∗) (*map succ arr1*) (*reverse arr2*): the extra annotations are guaranteed not to change its meaning.

We believe this gives a compelling approach to developing in-place array algorithms:

- Start with a clear, correct, functional algorithm.
- Tell the compiler how the output array should be populated in order for in-place execution to work.

When in-place algorithms are defined imperatively, getting the order of operations right is delicate and a major source of bugs. With our approach, the order of operations is cleanly separated from the semantics of the function, and is guaranteed to introduce no bugs into the code.

The rest of the paper describes the technical details of our design. We start by describing the intermediate imperative language which co-Feldspar compiles to (Section 2), which also supports the safe re-use of storage, and how to check that storage is safely re-used (Sections 3–4). We then present co-Feldspar itself (Section 5), the user-facing combinators for storage re-use, and how they are compiled to the intermediate language. We validate our method by evaluating an in-place FFT (Section 6) and finally describe the implementation of co-Feldspar in more detail (Section 7).

## 2 AN IMPERATIVE LANGUAGE FOR VERIFICATION

We do not attempt to check the safety of co-Feldspar programs at the source level. Instead, we first compile them into imperative code and then verify the safety of that code.

The intermediate imperative language is mostly very ordinary and provides arrays, loops, assertions and so on. It also adds one, crucial, new feature: *virtual copying*.

Virtual copying is the mechanism that allows us to express safe re-use of storage. Given an array *arr1*, the statement *arr2* := vcopy(*arr1*) creates an array *arr2* that gives the illusion of being a copy of *arr1*. Semantically, it is as if we had written:

```
arr2 := new int[len(arr1)];
for i := 0 to len(arr1) − 1 do arr2[i] := arr1[i]
```

However, *arr2* := vcopy(*arr1*) does not really make a copy of *arr1*. Rather, it makes *arr2* be an *alias* of *arr1*, a second pointer to the same underlying array. The compiler makes sure that the program behaves the same as if a real copy had been made.

To adapt the example from the introduction to work in place, writing its result to *arr1*, we just replace the line *result* := **new** *int32*[*n*] with *result* := vcopy (*arr1*). This gives the following program:

```
result := vcopy(arr1);
for i := 0 to n − 1 do
  result[i] := (arr1[i] + 1) ∗ arr2[n − i − 1]
```

In this program, *result* and *arr1* refer to the same underlying storage, but the compiler checks that they can safely share storage—that the program would work the same if the vcopy were replaced with a physical copy. This is a whole-program property—in this case, the vcopy is safe provided that a) *arr1* is not used in the remainder of the program, and b) *arr1* and *arr2* do not share storage. In the next section we will see how to check safety.

Finally, note that **new** and vcopy *bind* their result, rather than assigning it. That is, they create a new variable rather than updating an existing variable. This is important for verification because it makes alias analysis trivial: given any array variable, we can easily see exactly which other arrays it shares its underlying storage with.

## 3 REDUCING SAFETY TO ASSERTION CHECKING

In this section we show how to reduce safety checking to a standard program verification problem. The next section describes how we solve the resulting verification problem.

We verify a whole, closed program, not a function with inputs and outputs. Let us wrap the running example to make it a complete program:

```
arr1 := new int[n];
arr2 := new int[n];
  -- code to populate arr1 and arr2 omitted
result := vcopy(arr1);
for i := 0 to n − 1 do
  result[i] := (arr1[i] + 1) ∗ arr2[n − i − 1];
for i := 0 to n − 1 do print(result[i])    -- consume the result
```

Recall that a vcopy creates an alias of an array but gives the illusion of copying the array. The compiler's job is to make sure that this illusion is preserved, by checking that the program behaves the same as if a real copy had been made. In fact, the illusion can only be broken in one specific case: if two arrays share the same storage, writing some index *idx* in one array and then reading index *idx* in the other will give the wrong result. All we have to do is check that this situation is impossible.

Imagine that we want to check that the illusion holds for a *single* value of *idx*, say *checkedidx*. We can do this by *transforming* the program in the following way:

- For each array *arr*, we introduce a new Boolean variable $readable_{arr}$. This variable is going to be true if it is safe to read *arr*[*checkedidx*].
- After we write to *arr* [*i*], we check if $i = checkedidx$. If it is, reading *arr*[*checkedidx*] is safe (it will return the value that was just written), but reading any of its aliases is unsafe. We therefore set $readable_{arr}$ to true, and set the readable flag of all *arr*'s aliases to false. (Recall from the previous section that we can determine an array's aliases syntactically.)
- Before we read *arr*[*i*], we check if $i = checkedidx$. If it is, we assert that $readable_{arr}$ is true. (If it is false, the illusion is not preserved: the value returned by *arr*[*i*] would not be taken from this array but from one of its aliases.)
- When creating an array, we set its readable flag to true. When copying an array, we copy the new array's readable flag from the old array's.

For the program above, only *arr1* and *result* are aliases. The transformation results in the following program:

```
arr1 := new int[n]; readable_arr1 := true;
arr2 := new int[n]; readable_arr2 := true;
  -- code to populate arr1 and arr2 omitted
result := vcopy (arr1); readable_result := readable_arr1;
for i := 0 to n − 1 do begin
  if i = checkedidx then assert readable_arr1;
  if n − i − 1 = checkedidx then assert readable_arr2;
  result[i] := (arr1[i] + 1) ∗ arr2[n − i − 1];
  if i = checkedxix then begin
    readable_result := true;
    readable_arr1 := false
  end
end;
for i := 0 to n − 1 do begin
  if i = checkedidx then assert readable_result;
  print(result[i])
end
```

Now we send this transformed program to a program verifier, and ask it to check that the generated assertions pass for *all* possible values of *checkedidx*. (To quantify over all values of *checkedidx*, we can for example make *checkedidx* be an extra argument to the program.) If the verifier reports that all assertions always pass, then the use of vcopy is safe. Otherwise, there may be a safety bug and the program is rejected. Once the transformed code is verified, it is discarded: only the *original* code is compiled to machine code for execution.

Let us work through the transformed program above. Initially, all the *readable* variables are true. At first, we are running loop iterations where $i < checkedidx$; all the *readable* variables remain true this whole time. At some point, we reach the loop iteration where $i = checkedidx$. The first assertions pass, and then $readable_{arr1}$ is set to false. This means that it is no longer allowed to read *arr1*[*checkedidx*]. However, from now on we have $i > checkedidx$ so none of the assertions are triggered. We can therefore see that all the assertions in this program pass, and the original use of vcopy was safe.

Let us see what happens if we introduce a safety error into the program above:

- If we change the final printing loop so that it prints *arr1*[*i*] instead of *result*[*i*], then the transformed program will check that $readable_{arr1}$ holds. This will fail at the iteration $i = checkedidx$ because, as discussed above, the first loop sets $readable_{arr1}$ to false.
- If we change the line *result* := vcopy (*arr1*) so that it reads *result* := vcopy (*arr2*), then the assertion will fail for all values of *checkedidx* less than $n / 2$. For example, in the case $checkedidx = 0$, the iteration $i = 0$ will set $readable_{arr2}$ to false. When $i = n - 1$ we have $n - i - 1 = checkedidx$ and so the line assert $readable_{arr2}$ will fail. The verifier will thus fail to prove that the assertions pass for all values of *checkedidx* and the program will be rejected.

# 4 VERIFYING THE SAFETY CONDITIONS

The job we have left is to verify that all the assertions in the transformed program pass. Of course, this is an undecidable problem; solving it in general requires reasoning about loop invariants. The approach we take is fairly standard and combines two techniques: *satisfiability modulo theories* (SMT) solvers for reasoning about straight-line code, and *predicate abstraction* [15] for discovering loop invariants.

SMT solvers such as Z3 [11] check if a first-order formula is satisfiable with respect to some logical theory. SMT solvers support many theories that are useful for program verification, such as arrays, bit vectors and integers. Formulas using these theories provide a rich modeling language that can be used to reason about for example bounds checks and bit-twiddling code.

We use an SMT solver to reason about straight-line code, and code where loop invariants have already been inferred. Our verifier maintains a *context*, which is a mapping from program variables

to symbolic SMT values. We then step through the program, using the SMT solver as a sort of souped-up symbolic evaluation engine. When we reach an assignment statement, we update the context; when we reach an if-then-else, we locally assert the value of the if-test in each branch; when we reach a loop, we ask the SMT solver to prove that the invariant is initially true and preserved by the loop body. This approach is quite standard and modelled on Flanagan and Qadeer [15].

We also use the SMT solver to optimise away assertions present in the program. When we reach an assertion, we evaluate the expression in the current context to get a Boolean formula, and ask the SMT solver to prove this formula true. If the SMT solver finds a proof, we remove the assertion, since it must always hold.

We use predicate abstraction to compute loop invariants. Predicate abstraction takes as input a set $S$ of "interesting" Boolean expressions. It computes the strongest invariant that can be expressed as a Boolean combination of the expressions in $S$. For example, to discover an invariant such as $i > 0 \land (j \leqslant i \Rightarrow arr[j] < arr[i])$, the set $S$ would need to contain the formulas $i > 0$, $j < i$ and $arr[j] < arr[i]$.

Predicate abstraction is useful because finding interesting Boolean expressions is often easier than finding a whole invariant. It is also modular: we can add Boolean expressions from many sources and let the predicate abstraction algorithm find the right way to combine them. Co-Feldspar allows the user to give "hints", Boolean expressions which are added to the set $S$, and verification of combinators is often simpler when the combinators define appropriate hints.

We have two automatic heuristics for populating the set $S$ of of Boolean expressions:

- When accessing an array $arr[\,i\,]$, the formulas $i \leqslant checkedidx$ and $i \geqslant checkedidx$ are added to $S$.
- For any array $arr$, the variable $readable_{arr}$ is added to $S$.

For our example program, these two heuristics add (among others) the expressions $i \leqslant checkedidx$ and $readable_{arr1}$, from which predicate abstraction finds the loop invariant $i \leqslant checkedidx \Rightarrow readable_{input}$ for the main loop of the program. This invariant is proved to hold, and the SMT solver is then able to verify the safety of the program. This process is entirely automatic.

## 4.1 Invariantless Verification for For-Loops

Consider the variant of our example where we store the result (safely) in *arr2*:

```
  -- initialisation of arr1, arr2 elided
result := vcopy (arr2);
for i := 0 to (n − 1) / 2 do begin
  x := arr2[ i ];
  y := arr2[ n − i − 1 ];
  result[ i ] := (arr1[ i ] + 1) ∗ y;
  result[ n − i − 1 ] := (arr1[ n − i − 1 ] + 1) ∗ x
end
```

Verifying the safety of this vcopy requires discovering a rather complex invariant (the reader may like to puzzle it out). As mentioned in the last section, the user can give "hints" for suitable invariants, but it is not practical to expect the user to give reasonable hints. It may be possible for the *pairwise* combinator to supply such a hint, but currently it does not.

Instead, we are able to prove the safety of this loop *without* finding a loop invariant. The idea is as follows. Suppose we are given a for-loop of the form

**for** $i$ := 1 **to** $n$ **do** $S$

and that, in this for-loop, there is an array read $arr[\,j\,]$ which we want to verify. Now, suppose that this array read is *unsafe*. This means that $readable_{arr}$ must be false during the unsafe loop iteration. Furthermore, one of the following must be true:

1. $readable_{arr}$ was false *before* starting to execute the for-loop;
2. Some iteration of the for-loop set $readable_{arr}$ to false, and *the same* iteration performed the unsafe read;
3. Some iteration of the for-loop set $readable_{arr}$ to false, and *a later* iteration performed the unsafe read.

If we can show that none of these three cases can occur, then the read $arr[\,j\,]$ must in fact be safe. We eliminate the first case by asking the SMT solver if $readable_{arr}$ is true at the current program point. We eliminate the second case by asking the SMT solver to verify that the following program does not trigger an unsafe read:

```
havoc;
assume (readable_arr);
S
```

where *havoc* is an operation that puts the SMT solver into an arbitrary state satisfying the loop invariant, and *assume* ($readable_{arr}$) allows the SMT solver to assume that $readable_{arr}$ is true.

The third case is the most complicated. We eliminate it by considering *two* loop iterations, not necessarily consecutive, and checking that it cannot be the case that the first iteration sets $readable_{arr}$ to false and the second iteration triggers an unsafe read. This amounts to checking that the following program does not trigger an unsafe read:

```
  -- First put the program into an arbitrary state
  -- satisfying the loop invariant
havoc;
old_i := i;
  -- Assume that readable_arr is initially true
  -- but made false by the loop body
assume (readable_arr);
S;
assume (¬ readable_arr);
  -- Now go into a later loop iteration, in which readable_arr must
  -- still be false and the loop counter must have increased
havoc;
assume (¬ readable_arr);
assume (i > old_i);
S
```

If all three verification problems pass, the read in question is safe. This technique works well for loops where distinct iterations read and write to disjoint sets of array indices. In particular, it works for the example above. The SMT solver reasons as follows for the three cases:

(1) $readable_{arr}$ cannot be false before executing the loop.
(2) In each loop iteration, the reads to *arr2* come before the writes, so this cannot fail.
(3) From the loop invariant we know that $0 \leqslant i < n/2$. Hence if $i < j$ then the sets $\{i, n - i - 1\}$ and $\{j, n - j - 1\}$ are disjoint, and it is not possible to have a write to $result[\,checkedidx\,]$ followed by a read from $arr2[\,checkedidx\,]$.

Hence the SMT solver verifies the safety of the vcopy here. Note that verifying safety requires reasoning about arithmetic to see that distinct loop iterations work on disjoint parts of the array. The fact that the SMT solver understands arithmetic makes this problem possible to verify. The same method proves safety of the FFT example in Section 6, but there the SMT solver needs to use its knowledge of bitvectors to understand the bit-twiddling that goes on in the array indexing.

### 4.2 Static Verification of Assertions

In Feldspar, array accesses generate bounds checks, which are expressed as assertions in the generated imperative code. The user can also add their own assertions, which can be useful to express function preconditions or invariants. The program verifier tries to prove each assertion it comes across; if it can prove the assertion, it removes the assertion. Assertions left in the generated code that are not function preconditions often indicate potential runtime errors.

## 5 CO-FELDSPAR

In this section we give a brief overview of the co-Feldspar language, in which we have implemented our verifier. co-Feldspar is a domain specific language for the design of digital signal processing algorithms, embedded in Haskell and capable of generating code in C. Its roots in signal processing are reflected by the fact that it is an array programming language at its core: it is deliberately minimal and based on a low-level, but functional, representation of imperative programs. Although it is small, its core permits the addition of higher-level interfaces.

Statements in co-Feldspar are terms in its `Prog` monad, whereas expressions are pure; operations with side effects, such as those utilizing memory access, must take place at the statement level in the context of `Prog`. Memory is manipulated via non-nullable references that, in the case of arrays, are created, read and written using `newArr`, `getArr`, and `setArr` statements:

```
newArr :: Exp Length → Prog (Arr a)
getArr :: Arr a → Exp Ix → Prog (Exp a)
setArr :: Arr a → Exp Ix → Exp a → Prog ()
```

Where `Exp` is some expression type—as our focus will be on the verification of statements, we simply assume `Exp` supports whatever functionality we require.

As an example, the following function generates a program snippet that modifies a specific value in a given array:

```
inc :: Arr Int32 → Exp Ix → Prog ()
inc arr ix = do
  val ← getArr arr ix
  setArr arr ix (val + 1)
```

To compile `inc`, we first wrap it in a program that reads the index and array length from standard input. This wrapper is then given to the compiler's `icompile`:

```
icompile :: Prog () → IO ()
```

which generates and prints the following C (some variable declarations are omitted for brevity):

```
int main() {
  fscanf(stdin, "%u", &v0);
  fscanf(stdin, "%u", &v1);
  uint32_t _a2[v0];
  uint32_t *a2 = _a2;
  assert(v1 < v0 && "getArr out of bounds.");
  v3 = a2[v1];
  assert(v1 < v0 && "setArr out of bounds.");
  a2[v1] = v3 + 1;
  ...
}
```

Note the two assertions generated in the above C for each of the two potentially unsafe array operations: the correctness of `inc` depends on the length of `arr` being greater than the index `ix`. This condition is not explicitly stated by `inc` and the internal assertions for `getArr` and `setArr` are thus necessary to check that `inc` does not index `arr` out of bounds.

Conditions and assumptions can be introduced with the `assume`, which ensures that a program simply does not execute if its assumption is wrong, and `assert`, which are checked each time it is executed and, in contrast to assumptions, can be removed by the verifier if its condition is found to always hold. For example, adding the `assert (ix ⩽ length arr)` to the wrapper of `inc` ensures its index will stay within bounds and allow the verifier to discard the two internal assertions before any C is generated. The verifier is typically called with `iverify`, which attempts to statically verify assertions and invariants and prints the resulting program; if assertions cannot be verified, or a potential crash is detected, the assertions are left in place and errors are printed for whatever statements it found to be unsafe.

```
iverify :: Prog () → IO ()
```

Other than the mutable arrays presented so far, co-Feldspar also supports immutable arrays and includes statements for converting between mutable and immutable arrays as well as expressions for indexing into immutable arrays. Two such statements are provided: `unsafeFreeze` which turns a mutable array into a immutable one, and `unsafeThaw` which turns an immutable array back into a mutable one.

```
unsafeFreeze :: Arr a → Prog (IArr a)
unsafeThaw   :: IArr a → Prog (Arr a)
```

The above two statements are labeled as unsafe since no copies are created in the conversion—the converted array is actually a reference to the original array, with a new type. The intention is, of course, that the original array should not be mutated while the converted array is alive, and the co-Feldspar verifier will check that this is the case: `unsafeFreeze` and `unsafeThaw` are implemented directly using `vcopy`. (The "unsafe" epithet is historical,

since co-Feldspar originally had no safety verifier.) We can combine `unsafeFreeze` and `unsafeThaw` to create a user-level `vcopy`:

```
vcopy :: Arr a → Prog (Arr a)
vcopy = unsafeThaw . unsafeFreeze
```

## 5.1 Array Abstractions

As far as array computations go, the program type and its array instructions impose a fairly low-level, imperative style with explicit memory usage. In order to get back a higher-order compositional style of array programming, co-Feldspar provides *pull* arrays: a functional abstraction of arrays built on top of the core language that, as its name implies, excel at pulling out values from an array, and the operations it can efficiently implement are examples of this property. By the time the compiler is called, pull arrays are all but evaluated away and leave behind low-level programs with optimized loops. A pull array consists of a length and a function from indices to values:

```
data Pull a where
  Pull :: Exp Length → (Exp Ix → a) → Pull a
```

Pull arrays are notable for their non-recursive definition, which enables aggressive fusion; the composition of two pull arrays will not allocate any intermediate memory at run-time. As an example, consider a function which calculates differences between adjacent elements of a pull array:

```
diff (Pull l f) = Pull (l-1) (λi → f(i+1)-f(i))
```

Composing two or more applications of `diff` will result in a function that, once evaluated, returns a single pull array where all intermediate arrays have been eliminated:

```
   diff (diff (Pull l f))
⇒ diff (Pull (l-1) (λi → f(i+i)-f(i))
⇒ Pull (l-2) (λi → f(i+2)-f(i+1)-f(i+1)+f(i))
```

While fusing away intermediate arrays is often what we want when composing array functions, it is sometimes important to compute and store such arrays in memory, for example, to facilitate their sharing—even in `diff` there are two duplicate calls to `f(i+1)` as a result of fusion. Therefore, co-Feldspar provides the `manifest` function which stores a pull array in a given array:

```
manifest :: Arr a → Pull a → Prog (Arr a)
manifest arr (Pull len f) =
  for (0, 1, min (length arr) len) $ λix →
    setArr arr ix (f ix)
```

Using `manifest`, both `manifestFresh` and `manifestReuse` can be defined:

```
manifestFresh :: Pull a → Prog (Arr a)
manifestFresh pull = do
  arr ← newArray (length pull)
  manifest arr pull

manifestReuse :: Arr a → Pull a → Prog (Arr a)
manifestReuse arr (Pull len f) =
  copiedArr ← vcopy arr
  manifest copiedArr pull
```

Manifesting pull arrays into mutable ones allows us to arbitrarily split up an array computation into a sequence of intermediate arrays. As an example, consider the following program snippet:

```
vec :: Prog (Exp Int32)
vec = sum $ map (*2) $ (1...10)
```

First, `(...)` generates a pull array over all values in the range from one to ten. Then a mapping is applied with `map` to double the pull arrays' values before they are summed together with `sum` (these are pull array variants of the common list operations with similar names). We can split this computation into its three steps with a single array to hold each intermediate array:

```
vec :: Prog (Exp Int32)
vec = do
  arr ← newArr 10
  brr ← manifest arr $ (1...10)
  crr ← manifest arr $ map (*2) brr
  return $ sum crr
```

There are however cases where a single pull array is not enough to safely express a vector computation, even when intermediate arrays are manifested. For instance, consider the following example:

```
bad :: Prog (Exp Int32)
bad = do
  arr ← newArr 10
  brr ← manifest arr $ (1...10)
  crr ← manifest arr $ reverse brr
  return $ sum crr
```

First the array `arr` is allocated and initialized into `brr`, after which the reverse of `brr` is computed and also stored in `arr`. Given that reverse is implemented as:

```
reverse vec = Pull len (λi → vec ! len-i-1)
  where len = length vec
```

Manifesting `crr` into `arr` creates a for loop that will read indices out of `arr` that has already been modified, which leads to quite unexpected results.

co-Feldspar provides a safer alternative to the single array manifestation above with its double buffered storage type `Store`. A store is simply a pair of arrays, one marked as active and another as free, and each manifestation updates the active buffer and swaps it out for the free one. That way, a computation does not run the risk of overwriting the previous one. The price of a buffer is that it allocates twice as much memory as the single array solution.

Stores are defined and created as follows:

```
data Store a = Store (Arr a) (Arr a)

newStore :: Exp Length → Prog (Store a)
newStore l = Store <$> newArr l <*> newArr l
```

And the manifestation of pull arrays into stores is given by `store`:

```
store :: Manifestable vec ⇒ Store a → vec a →
  Prog (Arr a)
```

which writes the contents of a vector to its input buffer, swaps the buffers, and reads it back as an array, without making any copies. In contrast to the previous `manifest` function, `store` accepts any

Manifestable array, that is, any array type that can be stored in Arr array. Rewriting bad to make use of stores, instead of a single array, makes the program behave as expected:

```
good :: Prog (Exp Int32)
good = do
  st  ← newStore 10
  arr ← store st $ (1...10)
  brr ← store st $ reverse arr
  return $ sum brr
```

Not all cases where a single array is used for storage are however necessarily bad. In fact, they can be used to implement in-place updates if done correctly. Single arrays stores can be implemented for Store as well, we simply give it a single fresh array and a virtual copy of it:

```
newInPlaceStore :: Exp Length → Prog (Store a)
newInPlaceStore do
  arr ← newArr l
  brr ← vcopy arr
  return (Store arr brr)
```

As with every use of vcopy, inplaceStore creates a proof obligation that the store is used safely; it is not enough to replace the previous store in good with an in-place store, since reverse would exhibit the same problems as with a single array.

The problem is that reverse is simply not suited for in-place updates with its current implementation. For reversing arrays in-place, the common solution is to only traverse the array up to its half-way point and update elements at both the current index and the index at the opposite side simultaneously. This pattern is actually quite common and can be generalized into a form of pair-wise traversal over an immutable array, where each index consumes some two values, as long as the first value is at a smaller index than the second one. In co-Feldspar, the pairwise function implements the pattern:

```
pairwise :: Immutable vec ⇒
  (Exp Ix → (Exp Ix, Exp Ix)) →
  vec a → Prog (Arr a)
```

Which makes use of a function to determine which two values to consume for each index; the Immutable constraint accepts both immutable and pull arrays.

With in-place stores and the pair-wise traversal of arrays it is possible to define a in-place version of good as follows:

```
inplace :: Prog (Exp Int32)
inplace = do
  st  ← newInPlaceStore 10
  arr ← store st $ (1...10)
  brr ← store st $ reverse arr
  return $ sum brr
 where
  reverse = pairwise (λix → (ix, 10-ix-1))
```

Connecting inplace to a simple wrapper lets us verify the program as correct and generate the following C code (as before, definitions are omitted for brevity):

```
int main() {
  for (v1 = 0; v1 ⩽ 9; v1++)
    a0[v1] = v1 + 1;
  for (v2 = 1; v2 ⩽ 10 / 2; v2++) {
    r3 = a0[v2 - 1];
    r4 = a0[10 - (v2 - 1) - 1];
    a0[v2 - 1] = r3;
    a0[10 - (v2 - 1) - 1] = r4;
  }
  state5 = 0;
  for (v6 = 0; v6 < 10; v6++)
    state5 = a0[v6] + state5;
  ...
}
```

## 6 A CASE STUDY: FFT

The Discrete Fourier Transform (DFT) is specified as:

$$X_k = \sum_{j=0}^{n-1} x_j W_n^{jk}$$

where $W_n = e^{-2\pi i/n}$ is an $n^{th}$ root of unity. There are $n$ summations, each of $n$ elements, resulting in $O(n^2)$ complexity. Any algorithm that brings the complexity down to $O(n \log n)$ is known as a Fast Fourier Transform (FFT) and plays a central role in digital signal processing [12].

Cooley and Tukey's radix 2 Decimation in Frequency FFT (DIF) algorithm is one of the simplest and best known [10]. It consists of $n = \log N$ stages, each of which is made up of different arrangements of 2 input DFT components along with some multiplications by twiddle factors. A final permutation called bit reversal produces an output array identical to that produced by DFT. That one can make such a reduction in complexity is due to the rich algebraic properties of the $W_n^j$ terms—commonly referred to as twiddle factors, and the sharing of intermediate expressions. Various simple FFT implementations were explored in a paper about an early version of the Feldspar DSL [5], arriving at one that precomputed the twiddle factors, placing them in an array of length $N/2$ [6]. We rewrite the resulting FFT of that earlier exploration here, with the more explicit memory management of the newer co-Feldspar.

In the DIF algorithm, for input of length $N$, the even and odd-numbered parts of the output are each computed by a DFT with $N/2$ inputs. The inputs to those two half-sized DFTs can be computed by $N/2$ two-input DFTs. These smaller DFTs are then combined via butterfly networks, which themselves are size-2 DFTs pre-multiplied by twiddle factors. Nevertheless, a group of butterflies with length $2^{k+1}$ can be defined as follows:

```
bfly :: Immutable vec ⇒ Exp Ix → vec →
  Pull (Exp Complex)
bfly k as = Pull (length as) $ λi →
  let a = as ! i
      b = as ! flipBit i k
  in (testBit i k) ? (b-a) $ (a+b)

flipBit i k = k `xor` (1 .<<. (i2n i))
testBit a i = i2b (a .&. (1 .<<. i2n i))
```

For each index $i$, the $k^{th}$ bit of the indexed is used to determine if this output is to be given by an addition or a subtraction. The `Immutable` constraints is simply a collection of properties that immutable arrays display, such as supporting the pure indexing of `(!)`, and accepts both pull arrays and regular immutable arrays. `i2b` and `i2n` are used to convert an integer to a boolean and an integer to a floating-point number, respectively.

Multiplication by twiddle factors for an $n$ input FFT, which takes place only on the second half of the input, is again determined by the $k^{th}$ bit of index $i$ and defined as:

```
twids :: (Immutable tws, Immutable vec) ⇒
  tws → Exp Ix → Exp Ix → vec →
  Pull (Exp Complex)
twids tws n k vec = Pull (length vec) $ λi →
  let j = (leastBits k' i) .<<. (n'-1-k')
  in (testBit i k) ? (tws!j * vec!i) $ (vec!i)
  where
    n' = i2n n
    k' = i2n k

leastBits i a = a .&. complement (ones .<<. i)
```

Note that `twids` only handles the multiplication of twiddle factors with the input `vec`, the terms themselves are given through the parameter `tws`. By factoring out the terms we avoid unnecessary recomputation and can precompute the terms elsewhere.

The composition of `bfly` and `twids` makes out a step in the iterative DIT algorithm. To recover the bit-order of the original DFT algorithm we must perform a bit-reversal permutation of the final output. For blocks of length $2^k$, the bit-reversal will reverse the $k$ least significant bits of the binary representation of each index of the array, leaving all other bits alone. However, for the sake of clarity of our example, we will leave the output in its bit-reversed form and focus on the butterfly networks.

We define core of the FFT algorithm as a loop, where each a iteration takes one step in the iterative DIT algorithm:

```
fftCore st n tws vec =
  let step i = return . twids tws n i . bfly i
  in loopStore st (low,-1,0) (step . i2n) vec
  where low = i2n n - 1
```

The first argument is a double buffered store used to hold the intermediate arrays in each step of `loopStore`, a loop reminiscent of the standard for-loop but each iteration produces an update of an initial array and stores them in a buffer to avoid unnecessary copies:

```
loopStore :: Immutable vec ⇒ Store a → Range →
  (Exp Ix → Arr a → Prog (vec a)) → vec a →
  Prog (Arr a)
```

The second argument of `fftCore` is a vector of precomputed twiddle factors–as the twiddle factors don't change once computed they can be shared across multiple FFT runs as arguments. `n` is the number of stages needed in the FFT and `vec` the input vector, it is assumed that $l = 2^n$ where the length of `vec`.

`fftCore` is typically called for an input vector `vec` as follows:

```
fft :: Immutable vec ⇒ vec (Exp Complex) →
  Prog (Arr (Exp Complex))
fft vec =
  do st ← newStore (length vec)
     n  ← shareM (ilog2 (length vec))
     ts ← manifestFresh $
       Pull (twoTo (n-1)) (tw (twoTo n))
     fftCore st n ts vec

twiddle n k = polar 1 (-2 * pi * i2n k / i2n n)
twoTo   n   = 1 .<<. i2n n
```

That is, the first line allocates a new double-buffered store of the appropriate length, and the next line computes the number of stages using the integer base-2 logarithm. The third line precompute the twiddle factors, which is fed to `fftCore` in the fourth line. That is, `fft` takes care of an FFT's setup phase.

Here is the inner loop of the generated C (local variable declarations, assertions, and type casts are omitted for brevity):

```
for (v11 = 0; v11 ⩽ v0 - 1; v11++) {
  ...
  if (v11 & 1 << v10) {
    if (v11 & 1 << v10) {
      b13 = a2[v11 ^ 1 << v10] - a2[v11];
    } else {
      b13 = a2[v11] + a2[v11 ^ 1 << v10];
    }
    b12 = a6[(v11 & ~(4294967295 << v10))
               << v1 - 1 - v10] * b13;
  } else {
    if (v11 & 1 << v10) {
      b14 = a2[v11 ^ 1 << v10] - a2[v11];
    } else {
      b14 = a2[v11] + a2[v11 ^ 1 << v10];
    }
    b12 = b14;
  }
  a3[v11] = b12;
}
```

Note that the same conditional is repeated twice as a consequence of `testBit` being called twice, once in `twids` and once in `bfly`. Had we joined `twids` and `bfly` into one function this could have been avoided, but most clever C compilers will optimize away the inner if-statements regardless.

A similar FFT to the above one has been benchmarked previously and the generated C was found to perform reasonably well [3]. In order to achieve its speed, however, the compiler was explicitly told to not generate any assertions—sacrificing safety for performance. We repeat the earlier benchmarks here but instead rely on the verifier to safely improve the performance.

We start by replacing the out-of-place store with an in-place one: the butterfly networks, and subsequent multiplication with twiddle factors, form a bijective mapping that can be structured such that no input is read after its output has been set. That is, for a `step` at size `i`, if we consider for each index `k` together with `flipBit k i` then `pairwise` can be used to consume the inputs.

To implement the change we first replace the double-buffered store with an in-place one:

```
fft vec = do
  st ← newInPlaceStore (length vec)
  ...
```

The introduction of `pairwise` is then straightforward:

```
fftCore st n tws vec =
  let step i =
      return
    . pairwise (λk → (insertZero i k,
                      insertOne i k)
    . twids tws n i
    . bfly i
  ...
```

where `insertZero i k` inserts a 0 at bit $i$ of the binary representation of $k$, and `insertOne i k` inserts a 1.

We also add two assertions to `fft` that ensure that the input array's length is a power of two and that it has at least two elements:

```
fft vec =
  let len = length vec
  assert (1 `shiftL` n == len) "length is 2^n"
  assert (len ⩾ 2) "length is large enough"
  st ← newInPlaceStore (length vec)
  ...
```

The verifier automatically proves the safety of the in-place store, and removes all generated bounds check assertions from the FFT. To do so, it makes essential use of the two assertions: the bounds checks fail if the array size is not a power of two or less than two. The verifier took about 10 minutes to run, but we believe it can be made much faster by careful engineering.

The program used to benchmark the FFT is built on the same principles advocated by FFTW [16]: twiddle factors and number of stages are precomputed before the measurement starts, the input data is complex data in interleaved format, i.e., as arrays of complex numbers. To report FFT performance, the *mflops* of each FFT is plotted, which is a scaled version of the speed, defined by: mflops $= 5 * N * log_2(N)/T$, where $N$ is the number of data points and $T$ the time for one execution of the FFT in microseconds. The results of our benchmarks are given in figures 1 and 2. Each line represents one variant of the FFT and is identified by a two-character string consisting of in-place/out-of-place ($I$/$O$) and assertion removed/assertions retained ($A$/$N$). For example, $IA$ denotes the speed of our in-place FFT when the verifier has not been permitted to remove provable assertions. Figure 1 shows the raw speed of each version, while figure 2 shows the speed relative to $IN$.

For smaller arrays, there is no big difference between the in-place and out-of-place FFTs, but eliminating assertions increases speed by about 15%. This indicates that `gcc` was not able to eliminate all bounds checks by itself. This is not surprising, since to discharge them the compiler must reason about bit-level properties of numbers and use the fact that the array size is a power of two.

As the input array size approaches the machine's L2 cache size, which was 8 MB, running the FFT in place becomes more important:

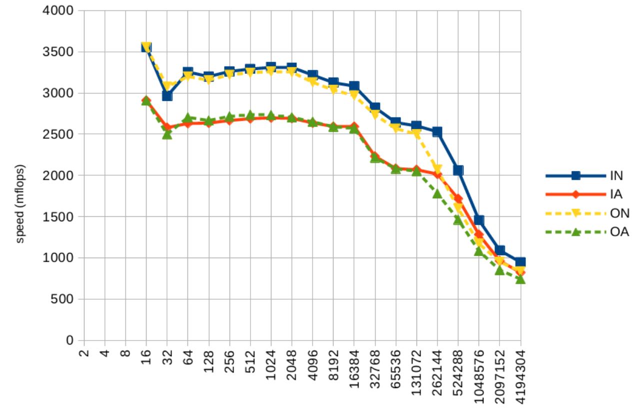


**Figure 1: double-precision complex transforms ($x$-axis shows size of input array)**

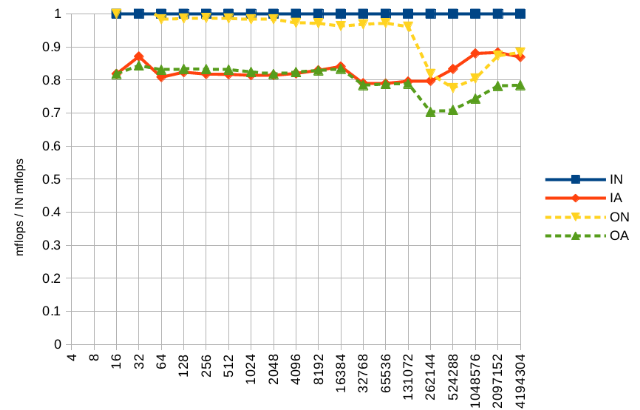


**Figure 2: relative performance of FFT variants ($x$-axis shows size of input array)**

the out-of-place FFT suddenly drops to 80% of the speed of the in-place FFT. The difference between $IN$ and $OA$ is almost 30%. As the array size becomes bigger still, both versions of the FFT suffer from cache pressure, but the in-place version is still faster.

## 7 IMPLEMENTATION

Our copy verification is built on top of co-Feldspar, which in turn is embedded in Haskell. co-Feldspar relies on Haskell to parse, typecheck, and desugar programs into its internal representation, which we then traverse to generate first-order logic and invoke a theorem prover for. This internal representation of co-Feldspar is a model of imperative programs. The general idea is that the monadic programs we write in Haskell can be viewed as sequences of primitive instructions, which makes their translation into first-order logic straightforward.

### 7.1 Programs

At the core of co-Feldspar is an abstract data type called `Program`, which captures imperative programs as monadic computations [1,

14]. Programs in co-Feldspar are parameterized on the primitive instructions that make out their statements. The instructions are themselves higher-order functors and can be parameterized on the programs they are part of to facilitate instructions with complex control flow. Programs are implemented as follows:

```
data Program i a where
  Return :: a → Program i a
  Instr  :: i (Program i) a → Program i a
  Bind   :: Program i a → (a → Program i b)
         → Program i b
```

The gist of the idea behind separating the instructions of programs from its monadic constructs is that an instruction's effect will only depend on its interaction with other instructions, which permits their sequencing to be handled separately. Also, with the aid of data types à la carte [22] and its type compositions operator (:+:), defining a instruction set for a program can be further seperated into individual instructions. As an example, we define part of the array language from section 2:

```
data Array m a where
  NewArr :: Exp Length → Array m (Arr a)
  GetArr :: Arr a → Exp Ix → Array m (Val a)
  SetArr :: Arr a → Exp Ix → Exp a → Array m ()

data IArray m a where
  UnsafeFreeze :: Arr a → IArray m (IArr a)
  UnsafeThaw :: IArr a → IArray m (Arr a)

data Control m a where
  Ife :: Exp Bool → m () → m () → Control m ()
  For :: Range → (Val Ix → m ()) → Control m ()

data Assert m a where
  Assert :: Maybe (Exp Bool) → Assert m ()
  Assume :: Exp Bool → Assert m ()
```

Here Exp is some expression type, and Range is a tuple of the for loop's lower bound, step size, and higher bound, in that order. As before, we focus on the statements and simply assume Exp supports whatever functionality we might need.

Array and IArray provide primitives for the management of mutable and immutable arrays. Control and Assert include primitives for control statements and assertions. Note that Val play a particular role: it is used whenever a value is produced by an instruction and represents a value in *any* expression—an abstract representation of values makes it easier to reuse instructions for verification and ensures that noone, other than built in conditionals, can pattern-match on values. Arr and IArr are used for similar reasons but represents mutable and immutable arrays.

We define our array language by putting these instructions together and instantiating a program with them:

```
type Ins = Array :+: IArray :+: Control :+: Assert

type Prog = Program Ins
```

Functions for adding statements to a program are then defined by injecting a statement from one of the smaller sets into a program:

```
newArr :: Exp Length → Prog (Arr a)
newArr = Instr . inj . NewArr
```

Code is generated from Prog by specifying the operational semantics of each instruction in Ins and interpreting Prog [4]. For the purpose of simplicity, however, consider the program type and its instructions as compiled code, even though a few more steps are actually required to run a program on a machine.

## 7.2 Verification

The Program type is somewhat unorthodox in that it includes a bind, since it is difficult to analyze the part of a program that is hidden behind its function. We therefore flatten a program into a sequence of instructions without binders before it is verified—a flat representation ensures a program's syntax tree is static during verification and allows our verifier to view the entire tree. The sequence type is defined as follows:

```
data Sequence ins a where
  Val :: a → Sequence ins a
  Seq :: b → ins (Sequence ins) b →
    Sequence ins a → Sequence ins a
```

Val lifts values into the sequence, and Seq takes a name and the instruction it binds and a sequence of previous instructions.

Programs are transformed into sequences through a traversal where embedded values are kept and binds are flattened and turned into a sequential composition. Any instruction encountered during this traversal are themselves converted into a first-order form, flattening or removing any higher-order functions and programs it contains, if any. For example, Control from section 7.1 uses a function to model the body of a for loop, and is thus converted into a equivalent, but first-order, form when lowered:

```
data FControl inv m a where
  FFor :: Maybe inv → Range → Val Ix → m () →
    FControl inv m ()
  ...
```

Note that not only did we flatten the function, but FControl is also parameterized with a loop invariant; if a loop invariant is added to a for loop, it is required to hold in the beginning of each iteration. The remaining instructions from section 7.1 can be reused for sequences:

```
type FIns = FControl :+: Array :+: IArray ...

type Seq = Sequence FIns
```

The verification of a sequence of instructions like Seq is reminiscent of symbolic execution, except that an SMT solver is used to do the symbolic reasoning: the state of a program is modeled as the state of the SMT solver plus a context, which is a mapping from variable names to SMT values. Symbolically executing a statement modifies this state to become the state after executing the statement. Typically, this modifies the context—when a variable has changed—or adds new axioms to the SMT solver.

Verification is performed within a monad that facilitates the manipulation of the underlying SMT solver and its context. The monad supports branching on the value of a formula, executing one branch if the formula is true and the other if the formula is false.

It also takes care of merging the contexts from the two branches afterwards, as well as making sure that any axiom added inside a branch is only assumed conditionally. The monad is based on a combination of the reader, writer, and state monads with the following components:

```
type Verify = RWST
  ([SMTExp], Mode)
  ([SMTExp], [Warn], [Hint], [Name])
  Context
  SMT
```

Where SMTExp is the expression type of the underlying SMT solver and SMT its monad type.

The reader monad in the Verify monad stack takes a list of formulas that are true in the current branch and a mode setting which determines whether the verifier should try to prove anything or just evaluate programs. The writer monad takes a disjunction which is true if the program has called break to exit the surrounding loop and lists for the warnings, hints, and names generated to detect, for example, a read of an uninitialized reference. The state monad takes the context—a map from variables to SMT values.

As an example, the verification of assertions will evaluate its condition within the current context of the monad and attempt to prove that it holds. If the condition can be shown to hold, the assertion is marked as unnecessary. On the other hand, if the condition could be falsified, the condition is kept and is assumed for the remaining verification. Such proof attempts are relayed to the verifier through prove, and the whole verification of assertions is defined as:

```
vAssert :: Assert m a → a → Verify (Assert m a)
vAssert (Assert (Just cond)) () = do
  bool ← toSMT <$> eval cond
  ok   ← prove bool
  if ok then do
    return (Assert Nothing)
  else do
    assume bool
    return (Assert (Just cond))
```

Instructions that manage data types, either references or arrays, are verified by mapping its type to a corresponding representation in SMT types. In the case of arrays, two such types are used: ArrCont and ArrBind. The first type, ArrCont, consists of two SMT expressions that hold values for the array and its bounds. The second type, ArrBind, consists of a name, for the source of the array, and three SMT expressions that keep track of whether the array is currently accessible, readable, and if it has previously been accessed in an unsafe way. It is during the verification of NewArr that these types are initialized and added to the context:

```
vArray :: Array m a → a → Verify (Array m a)
vArray (NewArr len) name = do
  lenE ← eval len
  var  ← fresh name
  poke name (ArrCont var lenE)
  poke name (ArrBind name true true false)
```

Where poke writes a type to the context under a given name, and fresh generates a new name.

The context and its two array types are queried whenever the verifier needs to check if an array update is safe or not. For instance, before an array value can be read the verifier must check that the array is accessible and that the given index is safe to read. If these conditions are not met, the array, and potentially its aliases, are marked as having been accessed in an unsafe manner. This procedure is implemented as:

```
vArray (GetArr arr ix) name = do
  ixE ← eval ix
  arr ← peek name
  ok  ← prove (accessible arr)
  if ok then do
    ok ← prove $
      not (ix .==. skolemIx) .||. readable arr
    unless ok $ do
      warn "unsafe to read as frozen array"
      unsafe arr
  else do
    warn "unsafe use of inaccessible array"
    mapM_ unsafe (aliases arr)
  hint ixE
  poke name (ValueBind (value arr .!. ixE))
```

The second proof attempt, for skolemIx, performs the vcopy index verification outlined in section 3. If any of the two proof attempts fails, the array and possibly ins aliases are marked as unsafe with unsafe, which sets the array's unsafe access flag to true. The verification of the remaining instructions are implemented in a similar manner.

Recall that verification is not only meant to identify unsafe indexing into virtual array copies, but also to improve the original program, as in the case of assertions that have been marked as unnecessary. A re-functionalization is therefore included, which turns our now improved first-order sequences back into higher-order programs, which can then be included in other programs and developed further. The verification of individual instructions, such as the above vArray function, are put together into a verifier for the instruction set FIns by a type class that distributes over data types à la carte type compositions operator (:+:).

## 8 RELATED WORK

A common approach for handling in-place updates in functional languages is linear types [23]. Values belonging to a linear type must used exactly once, they cannot be duplicated or destroyed. Such values can safely admit in-place array updates, and have inspired the reference ownership typing of Rust. Linear Haskell [8] is an extension of Haskell's typing system to include linear types where every function arrow can be declared linear to ensure functions and constructors really only do consume its argument once. Although linear types can make sure that an *entire* array is never read after being destroyed, it is not able to reason about individual array indexes, as this paper does.

Another language we should a mention is Futhark [17], a purely functional data-parallel array language aimed at GPU programming. Futhark employs a simple type system based on linear types that,

together with a restricted language of primitives, facilitates in-place updates while still supporting equational reasoning about functions and referential transparency. Combinators such as `map` are assigned linear types, but there is no support for checking that a custom array operation can be executed in place.

Rust [19] is another safe C language for developing reliable and efficient systems. Rust's static and affine type system is safe and expressive and prevents pointer aliasing errors as well as providing strong guarantees about isolation, concurrency, and memory safety. Its new type system enforces safe use of heap-based data structures, hygienic macros, and reference counting garbage collection as a library. Rust has a framework for property based testing, but lacks support for verification and static analysis of its programs. Spark/Ada [7] is language similar to Rust and provides high-assurance embedded programming, with a contract language and verification tools to prove invariants. While the language is programmed at a higher-level than typical C, it is also more restrictive than Rust, in particular, there are no references in Spark.

*Liquid Types* [20] is a call-by-value functional language with a similar goal to ours for verification. They offer a type system that combines the classical Hindly-Miller type inference with predicate abstraction to automatically infer dependent types that can be used to prove safety properties for its programs. Functions are given the usual types together with a *refinement* in a fixed language of predicates and the extracted conditions are discharged by an SMT-solver. Because the language of predicates is fixed, predicate abstraction can accurately infer precise refinements, and hence the burden for writing refinements on the programmer is quite low. This language of predicates is however fixed, whereas our verification method allows us full access to SMT theories at the cost of some overhead for translating the statement. The other major difference is that Liquid Types verify a single function at a time, whereas our approach verifies the whole program. This means that we get less precise error messages, but are able to verify code whose correctness argument is nonmodular, important for in-place algorithms.

Another, rather different, approach to refinement types is Dminor [9], which targets first-order functional languages instead and combines the ideas of refinement types with a type-test—a boolean expression testing whether a value belongs to a type. Their idea is to allow refinements to be written in the same programming language as programs are written; they formulate a semantic in which expressions denote terms and types are interpreted as first-order formulas. Expressions are however required to be pure, that is, they must all terminate, and have a unique denotation; expressions cannot depend on the current state of the store. Verification conditions are derived and discharged automatically with the help of Z3 [11]. The F* [21] language is similar in spirit to Dminor, as it utilizes value dependent types to guide an automatic extraction of verification conditions that discharged using theorem provers.

Ivory [13] is another language that is also similar to ours, and enforces memory safety and tries to avoid most undefined behavior while still providing low-level memory control. Ivory is embedded in Haskell and capable of expressing pre- and post-conditions, it can also emit run-time assertions to enforce them or use a model checking back-end to statically verify that they hold. Type-level regions are used to ensure memory references do not persist beyond the scope of their containing region.

The general idea of safe languages is not exclusive to functional languages and Cyclone [18] has done pioneering work in order to create a safe dialect of C. Cyclone is designed from the ground up to prevent the buffer overflows and memory management errors that are common in C programs, while retaining C's syntax and semantics. Most of Cyclone's language design indeed comes directly from C; Cyclone uses the C pre-processor, and, with few exceptions, follows C's lexical conventions and grammar. In contrast to co-Feldspar, Cyclone does not provide macro-programming facilities beyond its C pre-processor. The memory safety of Cyclone is verified by both static analysis and run-time checks, dynamic garbage collection is also offered with the help of memory regions.

## 9 CONCLUSIONS AND FUTURE WORK

We have presented an approach for improving the efficiency of functional array languages without sacrificing safety. The approach uses virtual copies of arrays to express in-place array transformations functionally, and a program verification engine to make sure that the virtual copies are used safely. The programmer can define array algorithms at a high level of abstraction and compile them down to efficient in-place code with mutations and minimal bounds checks. In the case study we performed, the combination of in-place updates and bounds check removal increased performance by 30% in some cases, without harming safety.

By using an embedded domain specific language co-Feldspar, and Haskell's type classes, we retain the elegance and modularity of traditional functional array programming. We are mainly interested in using co-Feldspar as the source language; however, the general idea of virtual copies and their verification is not dependent on co-Feldspar, and so may be of interest to other developers of array languages.

We would like to improve upon the generation of invariant hints for the solver, in order to speed up the verification of larger examples with a complex control structure. By evaluating virtual copies on other signal processing algorithms, we plan to find useful in-place execution strategies along the lines of `pairwise`. We would also like to explore the use of virtual arrays as an automatic optimisation, for example, turning copies and allocations into virtual copies where safe.

## ACKNOWLEDGMENTS

The authors acknowledge the contributions to this work made by Emil Axelsson and Anders Persson in the development of Feldspar and its FFT library. This work was supported by the Swedish Research Council (VR) grant 2016-06204, *Systematic testing of cyber-physical systems (SyTeC)*.

## REFERENCES

[1] Heinrich Apfelmus. 2010. The Operational monad tutorial. *The Monad. Reader* 15 (2010), 37–55.
[2] Markus Aronsson and Mary Sheeran. 2017. Hardware software co-design in Haskell. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 162–173.
[3] Emil Axelsson. 2016. Benchmarking FFT in Feldspar. http://fun-discoveries.blogspot.com/2016/11/benchmarking-fft-in-feldspar.html
[4] Emil Axelsson. 2016. Compilation as a Typed EDSL-to-EDSL Transformation (Blog post). http://fun-discoveries.blogspot.se/2016/03/.

[5] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2010. The design and implementation of Feldspar. In *Symposium on Implementation and Application of Functional Languages*. Springer, 121–136.

[6] Emil Axelsson and Mary Sheeran. 2011. Feldspar: Application and implementation. In *Central European Functional Programming School*. Springer, 402–439.

[7] John Gilbert Presslie Barnes. 2003. *High integrity software: the spark approach to safety and security: sample chapters*. Pearson Education.

[8] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 5.

[9] Gavin M Bierman, Andrew D Gordon, Cătălin Hriţcu, and David Langworthy. 2010. Semantic subtyping with an SMT solver. In *ACM Sigplan Notices*, Vol. 45. ACM, 105–116.

[10] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] Pierre Duhamel and Martin Vetterli. 1990. Fast Fourier transforms: a tutorial review and a state of the art. *Signal processing* 19, 4 (1990), 259–299.

[13] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt free ivory. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 189–200.

[14] Emil Axelsson. [n. d.]. A version of Operational suitable for extensible EDSLs. http://hackage.haskell.org/package/operational-alacarte

[15] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. ACM. https://www.microsoft.com/en-us/research/publication/predicate-abstraction-software-verification/

[16] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *ACM SIGPLAN Notices* 52, 6 (2017), 556–571.

[18] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[19] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[20] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 159–169.

[21] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 266–278.

[22] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

[23] Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. Citeseer, 347–359.