Recursive Array Comprehensions in a Call-by-Value Language

Artjoms Šinkarovs
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Sven-Bodo Scholz
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

Robert Stewart
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
r.stewart@hw.ac.uk

Hans-Nikolai Vießmann
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
hv15@hw.ac.uk

ABSTRACT

Recursive value definitions in the context of functional programming languages that are based on a call-by-value semantics are known to be challenging. A lot of prior work exists in the context of languages such as Scheme and OCaml.

In this paper, we look at the problem of recursive array definitions within a call-by-value setting. We propose a solution that enables recursive array definitions as long as there are no cyclic dependences between array elements. The paper provides a formal semantics definition, sketches possible compiler implementations and relates to a prototypical implementation of an interpreter in OCaml. Furthermore, we briefly discuss how this approach could be extended to other data structures and how it could serve as a basis to further extend mutually recursive value definitions in a call-by-value setting in general.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; **Recursion**; **Compilers**; • **Computing methodologies** → *Parallel programming languages*;

KEYWORDS

Functional languages, Array programming, Call by value, Array comprehensions

ACM Reference Format:
Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Stewart, and Hans-Nikolai Vießmann. 2017. Recursive Array Comprehensions in a Call-by-Value Language. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205368.3205373

IFL 2017, August 30-September 1, 2017, Bristol, United Kingdom

ACM ISBN 978-1-4503-6343-3/17/08.
https://doi.org/10.1145/3205368.3205373

1 INTRODUCTION

The use of arrays for parallel computing has seen a renaissance over the last decade. Array based languages have gained popularity not only in the mainstream (*e.g.* Matlab, R, Julia) but also in the realm of functional programming languages. As demonstrated by languages such as SaC [25], DpH [6], Accelerate [7], Futhark [14], Feldspar [4], and others, array operations can be embedded into a functional setting in a way that combines nicely high-level abstractions with ample opportunities for program optimisations and code generation of high-performance codes for parallel architectures.

The core of functional array languages is typically built around a small set of array generating skeletons. By and large, these are variants of map, reduce or scan combinators that relate generators of index sets to computations of corresponding array elements. We refer to these generically with the term *array comprehensions*.

Array comprehensions enable specifications very close to their mathematical counterparts. As an example, consider transposition of a matrix a and its formulation using SaC language array comprehensions:

$$b_{ij} = a_{ji} \qquad \texttt{b = \{ [i,j] -> a[j,i] \}}$$

Similarly, for matrix multiplication we have:

$$c_{ij} = \sum_{k=1}^{n} (a_{ik} b_{kj}) \quad \texttt{c = \{ [i,j] -> sum (a[i,.] * b[.,j]) \}}$$

As long as array expressions do not refer to the elements they are defining, translation from mathematical specification into array comprehensions is usually straight forward. In case of self-referential definitions some extra work is required. For example, consider Choleski Decomposition [11] that computes a matrix l from a matrix a using the following equation:

$$l_{ij} = \begin{cases} \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} & i = j \\ \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right) & i > j \\ 0 & i < j \end{cases}$$

Note that most of the element definitions refer to other elements of the matrix l itself. In a strict setting, this usually precludes a direct implementation through a single array comprehension. Instead, a programmer needs to identify and break recursive dependencies,

usually by composing array comprehensions that compute parts of the problem. Often, this is being done by expressing the computation as a suitably chosen sequential traversal over all array elements. For the above computation of the array *l*, a row-major traversal observes the required dependencies. In SaC this can be expressed as:

```
l = a;
for (i = 0; i < shape(a)[0]; i++) {
  for (j = 0; j < i+1; j++) {
    s = sum(take([j],l[i]) * take([j],l[j]));
    l[i,j] = (i == j) ? sqrt(a[i,i]-s)
           : (i >  j) ? (1.0/l[j,j]) * (a[i,j]-s)
           :            0.0;
  }
}
```

where `take ([j], l[i])` selects first *j* elements from a vector `l[i]`; and `sum` adds all the elements of its argument array. Note that the order of computation of all elements of *l* has been fixed. Explicit *for*-loops suggests an element by element computation.

We see two major disadvantages of such an approach. First, a programmer has to identify a suitable order that implements the given recursive specification. This is far from trivial in the general case.

Secondly, the order of computation chosen by a programmer might be overly restrictive, unnecessarily constraining concurrency of the original formulation. In our example, it would be possible to compute all elements at positions `l[i,0]` from the very beginning. Likewise, as soon as all elements `l[i,0]. . . l[i,j]` are computed, all elements `l[k,0]. . . l[k,j]` with `k>i` can be computed as well. While advanced compiler technology, *e.g.* the work in the context of the Polyhedral Model [10], can be leveraged to regain the concurrency that is inherent in the mathematical specification of the given example, in general, this is not always possible.

In this paper, we propose a way for enabling recursively defined array comprehensions within a strict array language. We envision specification of Cholesky decomposition to look like this:

```
letrec l = {
  [i,j] -> {
    s = sum(take([j],l[i]) * take([j],l[j]));
    (i == j) ? sqrt( a[i,i] - s)
  : (i >  j) ? (1.0/l[j,j]) * (a[i,j]-s)
  :            0.0
}}
```

The idea here is to introduce a scope for recursive definitions by means of a *letrec* construct. It ensures that all occurrences of the variable *l* on the right hand side of the assignment recursively refer to the array that is being defined. This program does not only match the mathematical specification directly but it also alleviates the programmer from identifying and correctly specifying dependencies, and it facilitates more parallelism to be directly available.

At this point, one might wonder whether switching from strict to lazy evaluation would not solve this problem immediately. While lazy evaluation would ensure that the above array comprehension delivers the desired result, a switch to lazy evaluation or at least lazy arrays in general would have several undesirable implications.

Firstly, it would very likely restrict the possible parallelism since, in a lazy setting, evaluations are triggered by demand, and demand does not usually come in parallel.

Secondly, a key for compiling compositions of array comprehensions into code whose performance levels are commensurate with, or superior to, those of hand-tuned imperative codes lies in a tight control when arrays are being materialised in memory and when they can be reused for updates or entirely new arrays. In most cases, this is achieved by a combination of strict evaluation (call by value) and some mechanism to enable updates in place, be it non-delayed garbage collection (reference counting) [12] or single-threaded use of arrays through mechanisms such as state monads [20] or uniqueness typing [5].

Strictness fosters parallelism in the evaluation of array elements and it ensures that references to arrays that contribute to the computation of a new array do not need to be kept alive due to delayed element computations. Additionally, some form of update in place mechanisms is crucial to reuse arrays that are no longer needed. In particular when dealing with large arrays, these aspects often turn out to be dominating the overall performance.

The contributions of this paper are:

- We suggest a mechanism for integrating recursive array comprehensions into a call-by-value setting. This mechanism is based on some form of temporal lazy evaluation but is designed to keep the need for persisting closures as low as possible.
- We provide a formal semantics for recursive array comprehensions in the setting of a small applied λ-calculus that can be seen as the core language of SaC or other functional array languages. An implementation of that semantics is provided in the form of an interpreter written in OCaml and available on GitHub[1].
- We discuss how a compiled code may look like, including aspects related to parallel executions.
- We provide an outline how this approach can be applied to extend other call-by-value languages' capabilities to deal with recursive value definitions.

2 CORE ARRAY LANGUAGE

To simplify our presentation, and to keep the proposed ideas language-agnostic, we introduce a minimalistic strict higher-order functional language which we use to present our techniques. It constitutes an applied λ-calculus extended by an array comprehension construct, called *imap* operator.

Any of the existing strict array-based languages, *e.g.* APL, SaC, Futhark can be mapped into our core language. Although we restrict ourselves to one-dimensional arrays, this is purely done for conciseness of the presentation. An extension to support multi-dimensional arrays can be done straight-forwardly. Syntactically different array comprehensions such as Futhark's *map*, SaC's *with-loop* or APL's *'Double-Dot' operator* can be expressed using our *imap* operator.

2.1 Language Syntax

We define the syntax for our minimalistic language as:

$$e \ ::= \ c \qquad \text{(constants)}$$

[1]The interpreter can be found at https://github.com/ashinkarov/heh. See Section 4.1 for more details.

	x	(variables)
|	$\lambda x.e$	(abstractions)
|	$e\ e$	(applications)
|	*if* e *then* e *else* e	(conditionals)
|	*letrec* $x = e$ *in* e	(recursive let)
|	$e + e, \ldots$	(built-in binary)
|	*imap* $e\ e$	(index map)
|	$e.e$	(selections)
c	::= d	
|	$[\, d, \ldots \,]$	(constant array)
d	::= $0, 1, \ldots$	(natural numbers)

As constants, we support only natural numbers and arrays of constants. Abstractions and applications use standard notation. Conditionals expect a natural number as predicate; 0 is considered *false* and any other number represents *true*. The *letrec* construct is a recursive let binding. Primitive operations are built-in and they are defined on scalars in the usual way. The key operations are the *imap* (index map) construct for defining immutable one-dimensional arrays, and the selection operation that provides access to individual array elements.

As a simple example for our *imap* contruct consider the following expression:

```
imap 5 λi.i = [0,1,2,3,4]
```

it has two arguments: the length of the resulting array and the mapping function from indexes to values. The function is called for the range of indexes from 0 to 5 (including 0, but excluding 5), and results of its application are put together at the corresponding array indices forming an array value. The above expression evaluates to an immutable array of 5 elements. We assume that all arrays are one-dimensional. Array indexing starts with 0.

Conditionals within the *imap* function, make it possible to partition the *imap* index-space:

```
imap 5 λi.if i < 2 then 1 else 2 = [1,1,2,2,2]
```

2.2 Language Core Rules

To give a meaning to programs we will use *natural semantics* [19]. We use the following values:

$$v ::= v_u \mid [v_u, \ldots, v_u] \qquad v_u ::= d \mid [\![\lambda x.e, \rho]\!]$$

where v_u can be a scalar constant or a function closure; $[v_u, \ldots, v_u]$ is a homogeneous array of numbers or functions; ρ is the environment. To make sharing more visible, instead of binding variables to values in the environment, we bind variables to *pointers*. Pointer-value bindings are kept in a *storage*, commonly denoted with S in this paper.

$$\rho ::= \cdot \mid \rho, x \mapsto p \qquad S ::= \cdot \mid S, p \mapsto v$$

Environment and storage look-ups happen *right to left* and are denoted as $\rho(x)$ and $S(p)$ respectively. Judgements take the following form:

$$S; \rho \vdash e \Downarrow S'; p$$

This means that within the storage S and the environment ρ we can show that e reduces to the storage S' and the pointer p. To improve readability we introduce a shortcut notation

$$S; \rho \vdash e \Downarrow S'; p \Rightarrow v \qquad \text{for} \qquad S; \rho \vdash e \Downarrow S'; p \wedge S'(p) = v$$

The core rules are:

$$\textsc{Const}\ \frac{S_1 = S, p \mapsto c}{S; \rho \vdash c \Downarrow S_1; p} \qquad \textsc{Var}\ \frac{x \in \rho \qquad \rho(x) \in S}{S; \rho \vdash x \vdash S; \rho(x)}$$

$$\textsc{Prf}\ \frac{\oplus \in \{+, -, \ldots\} \qquad S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow v_1 \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow v_2 \qquad S_3 = S_2, p \mapsto v_1 \text{ sem } (\oplus)\, v_2}{S; \rho \vdash e_1 \oplus e_2 \Downarrow S_3; p}$$

$$\textsc{Abs}\ \frac{S_1 = S, p \mapsto [\![\lambda x.e, \rho]\!]}{S; \rho \vdash \lambda x.e \Downarrow S_1; p} \qquad \textsc{App}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [\![\lambda x.e', \rho']\!] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \qquad S_2; \rho', x \mapsto p_2 \vdash e' \Downarrow S_3; p_3}{S; \rho \vdash e_1 e_2 \Downarrow S_3; p_3}$$

$$\textsc{If-true}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow n \in \mathbb{N}, n \neq 0 \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2}{S; \rho \vdash \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \Downarrow S_2; p_2}$$

$$\textsc{If-false}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow n \qquad n = 0 \qquad S_1; \rho \vdash e_3 \Downarrow S_2; p_3}{\rho \vdash \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \Downarrow S_2; p_3}$$

Here, 'sem $(\oplus)$' corresponds to the meaning of the binary operation. The rules for *imap* and selections are:

$$\textsc{Imap}\ \frac{S; \rho \vdash e_1 \Downarrow S'; p_n \Rightarrow n \in \mathbb{N} \qquad S'; \rho \vdash e_2 \Downarrow S_0; p_f \qquad \forall k \in \{0, \ldots, n-1\} : S_k; \rho, f \mapsto p_f \vdash f\ k \Downarrow S_{k+1}; p_k \Rightarrow v_k}{S; \rho \vdash \textit{imap } e_1\ e_2 \Downarrow S_n, p \mapsto [v_0, \ldots, v_{n-1}]; p}$$

$$\textsc{Sel}\ \frac{S; \rho \vdash e_1 \Downarrow S_1; p_1 \Rightarrow [v_0, \ldots, v_{n-1}] \qquad S_1; \rho \vdash e_2 \Downarrow S_2; p_2 \Rightarrow k \in \mathbb{N}, k < n}{S; \rho \vdash e_1.e_2 \Downarrow S_2, p \mapsto v_k}$$

The *imap* rule evaluates the length of the array and the mapping function, then it applies the mapping function to indices $\{0, \ldots, n-1\}$ and combines results into a vector. The selection rule extracts the k-th element from the array (where array indexing is zero-based).

The rules we have defined so far are sufficient to prove evaluation of non-recursive *imap*s. For example, we can show that

$$\cdot; \cdot \vdash \textbf{imap } 5\ \lambda i.i \Downarrow S; p \Rightarrow [0, 1, 2, 3, 4]$$

An interpreter for the presented language can be straight-forwardly derived from the presented rules.

2.3 Letrec Rule

The rule for the *letrec* given in [19] can be transliterated in our notation as follows:

$$\textsc{Letrec-Kahn}\ \frac{S, p \mapsto v; \rho, x \mapsto p \vdash e_1 \Downarrow S_1; p' \Rightarrow v \qquad S_1; \rho, x \mapsto p \vdash e_2 \Downarrow S_2; p_2}{S; \rho \vdash \textit{letrec } x = e_1 \textit{ in } e_2 \Downarrow S_2; p_2}$$

Before evaluating e_1 we need to guess the value that e_1 evaluates to. This means that the entire mechanics of recursive evaluation is not exposed. This makes the rule concise, but the previous claim that the interpreter is straight-forwardly deducible from the rules does not hold anymore. However, this rule makes it possible to prove that recursive *imap*s evaluate to arrays; for example:

```
·;· ⊢ letrec a = imap 5
                     λi.if i = 0 then 0
                        else a.(i−1)+1
      in a
⇓ S;p ⇒ [0, 1, 2, 3, 4]
```

Let us define a more explicit version of the letrec rule, yet preserving the above property. We start with a typical approach to letrec evaluation used in many strict languages. This can be denoted as follows:

$$\textsc{Letrec}\quad \frac{S_1 = S, p \mapsto \bot \qquad \rho_1 = \rho, x \mapsto p \qquad S_1;\rho_1 \vdash e_1 \Downarrow S_2;p_2 \qquad S_3 = S_2[p_2/p] \qquad S_3;\rho, x \mapsto p_2 \vdash e_2 \Downarrow S_4;p_4}{S;\rho \vdash \mathit{letrec}\ x = e_1\ \mathit{in}\ e_2 \Downarrow S_4;p_4}$$

The $\bot$ value ensures that if a letrec variable is accessed during the computation of e_1, as for example in the expression $\mathit{letrec}\, x = x\ \mathit{in}\, x$, the evaluation will fail. $S[p_2/p]$ denotes substitution of the $x \mapsto p$ bindings with $x \mapsto p_2$ inside of all enclosed environments, for all variables x that bind to p. Such a rule makes it possible to create the circular references that we need. Consider an example of evaluating the program $\mathit{letrec}\ f = \lambda x.f\ x\ \mathit{in}\ f$:

$\cdot;\cdot$	$\mathit{letrec}\ f = \lambda x.f\ x\ \mathit{in}\ f$	Letrec
$S = p \mapsto \bot; \rho = f \mapsto p$	$\lambda x.f\ x$	Abs
$S_1 = S, p_1 \mapsto [\![\lambda x.f\ x, f \mapsto p]\!];\cdot$	$\mathit{letrec}\ f = p_1\ \mathit{in}\ f$	$S_2 = S_1[p_1/p]$
$S_2 = S, p_1 \mapsto [\![\lambda x.f\ x, f \mapsto p_1]\!]; f \mapsto p_1$	f	Var
$S_2;\cdot$	p_1	□

As can be seen, the closure with the recursive function has the environment that correctly references the closure. Unfortunately, this rule does not treat recursive *imap*s properly. The reason for this is the inability to evaluate selections into *imap*s while their evaluation is in progress, even if the elements that are to be accessed have already been computed.

3 RECURSIVE IMAPS

To regain this ability, we can extend the above language and allow lazy evaluation of *imap*s. One way of doing this is to introduce a new value for an *imap* closure:

$$[\![\mathit{imap}\ p_n\ p_f, \{\}]\!]$$

which contains the *imap*, where both arguments are pre-evaluated and an index-value mapping to store a partial result. The partial result will be updated each time we make a selection into an *imap* closure. The rules to create such a closure and evaluate selections into it are:

$$\textsc{Imap-Lazy}\quad \frac{S;\rho \vdash e_1 \Downarrow S_1;p_1 \qquad S_1;\rho \vdash e_2 \Downarrow S_2;p_2 \qquad S_3 = S_2, p \mapsto [\![\mathit{imap}\ p_1\ p_2, \{\}]\!]}{S;\rho \vdash \mathit{imap}\ e_1\ e_2 \Downarrow S_3;p}$$

$$\textsc{Sel-Lazy-Imap-Full}\quad \frac{S;\rho \vdash e_1 \Downarrow S_1;p_1 \Rightarrow [\![\mathit{imap}\ p_1'\ p_2', M]\!] \qquad S_1;\rho \vdash e_2 \Downarrow S_2;p_2 \Rightarrow m \in \mathbb{N}, m < S_1(p_1') \qquad m \in M}{\rho \vdash e_1.e_2 \Downarrow S_2, p \mapsto M(m);p}$$

$$\textsc{Sel-Lazy-Imap-Empty}\quad \frac{S;\rho \vdash e_1 \Downarrow S_1;p_1 \Rightarrow [\![\mathit{imap}\ p_1'\ p_2', M]\!] \qquad S_1;\rho \vdash e_2 \Downarrow S_2;p_2 \Rightarrow m \in \mathbb{N}, m < S_1(p_1') \qquad m \notin M \qquad S_2;\rho, f \mapsto p_2' \vdash f\ m \Downarrow S_3;p_3 \Rightarrow v \qquad S_4 = S_3 \oplus_{p_1} [\![\mathit{imap}\ p_1'\ p_2', M \cup \{m \mapsto v\}]\!]}{\rho \vdash e_1.e_2 \Downarrow S_4, p \mapsto v;p}$$

The Imap-Lazy rule evaluates *imap* arguments and creates a closure, with an empty partial result. The Sel-Lazy-Imap-Full rule asserts that the value at the given index can be found within the partial result M of the *imap* closure. In this case we look-up the value in M. The Sel-Lazy-Imap-Empty rule asserts that the value is not within the mapping M, in which case we apply the *imap* function at the given index and we memoize the evaluated result. To do the memoization we update the storage using $S \oplus_p v$ operation that replaces $p \mapsto \_$ mapping with $p \mapsto v$ in S.

These extensions make it possible to handle recursive *imap*s, even if some of the elements are not defined:

```
letrec a = imap 5
                λi.if i = 0 then a.i
                   else 1
in a
```

That is, all the selections into *a* will deliver a result, except if we select at index 0. The price for this is the necessity to maintain and update partial results within *imap* closures. This suggests that we cannot arbitrarily swap Imap and Imap-Lazy, as such a choice has an implication on termination properties of our programs.

At this point, we have two sets of rules for *imap* and selections. How do we chose which one to use, assuming that we prefer strict evaluation over the non-strict one.

3.1 Approach

Our approach lies in minimizing the lifetime of *imap* closures. To do this, we restrict the way recursive *imap*s can be defined. A valid recursive *imap* shall be defined as:

```
letrec x = imap e_s e_f in e
```

For such cases we intend to create closures only in case e_f potentially references x at runtime. This can be approximated by checking whether x can be found in free variables of e_f. We denote this as $x \in FV(e_f)$. Note that this check does not imply that x *will* be referenced at runtime, as x can be a part of a sub-expression that discards x. As an example, consider the expression:

```
letrec x = imap 5 λi.if 0 then x else 1 in x
```

However, in case we statically know that $x \notin FV(e_f)$, the corresponding *imap* can be evaluated strictly.

While the aforementioned measure avoids the creation of *imap*-closures for all non-recursive definitions, all other cases will generate *imap*-closures and these will persist until either all elements of the array have been accessed at least once or no further references to the array exist anymore. To shorten the lifetime of *imap*-closures and, with them, the lifetime of references to all arrays that contribute to the computation of such closures, we insist that all elements

of such *imap*-closures are being evaluated as soon as the closures have been created. We achieve this using the following two rules.

Force

$$\frac{S_0(p) = \left[\!\left[imap\ p_s\ p_f\ M \right]\!\right] \qquad S_0(p_s) = n \qquad \forall i \in \{0,\ldots,n-1\} : S_i, p_i \mapsto i; \rho, a \mapsto p, ix \mapsto p_i \vdash a.ix \Downarrow S_{i+1}; p_i \Rightarrow v_i \qquad S_r = S_n, p_r \mapsto [v_0,\ldots,v_{n-1}]}{S_0; \rho \vdash force\ p \Downarrow S_r; p_r}$$

Letrec-1

$$\frac{x \in FV(e_1) \qquad S_1 = S, p \mapsto \bot \qquad \rho_1 = \rho, x \mapsto p \qquad S_1; \rho_1 \vdash e_s \Downarrow S_2; p_s \qquad S_2; \rho_1 \vdash e_f \Downarrow S_3; p_f \qquad S_3 \oplus_p \left[\!\left[imap\ p_s\ p_f\ \{\} \right]\!\right]; \rho_1 \vdash force\ p \Downarrow S_4; p' \qquad S_5 = S_4 \oplus_p S_4(p') \qquad S_5; \rho_1 \vdash e \Downarrow S_6; p_r}{S; \rho \vdash letrec\ x = imap\ e_s\ e_f\ in\ e \Downarrow S_6; p_r}$$

To simplify the notation we introduce a meta-operator called *force* which has a single argument — a pointer to an *imap* closure. The Force rule evaluates *force* by evaluating selections into the closure for all the elements in the given index space.

The Letrec-1 rule starts with evaluating shape[2] expression e_s and function expression e_f for the given *imap*. Note that letrec variable x is bound to p which is bound to $\bot$, which ensures that any immediate references to x will result in failure. When e_s and e_f are evaluated, the value of p is being updated with a fresh closure. After that all the values in this closure are forced via *force* operator, then the value of p is substituted with the value returned by *force*. Finally the goal expression of the letrec is evaluated.

Such a formulation guarantees that when we start evaluating the body of the letrec, any potential *imap* closure has been evaluated completely. In other words, the arrays are strictly evaluated again. Consequently, recursive definitions that contain circular dependencies will lead to non-termination, even if the corresponding elements are never being accessed. We will discuss the performance implications of this decision in the next section.

The final semantics of our language includes all the rules from Section 2, the rules for lazy selections (Sel-Lazy-Imap-Full and Sel-Lazy-Imap-Empty), and the two rules Force and Letrec-1. The rule Imap-Lazy is not included, as we want the only way to create an *imap* closure to be by means of Letrec-1.

4 IMPLEMENTATIONS

4.1 The Heh Interpreter

We demonstrate that the proposed idea works by implementing it in the Heh[3] programming language [27]. Heh is an interpreted language implemented in OCaml. The core features of Heh are very similar to the language we have presented in this paper. The main difference is that Heh natively supports multi-dimensional and infinite arrays, and evaluates all the *imap*s lazily.

First of all, we implement a flag `-finite-imap-strict` that makes Heh behave according to the semantics described in Section 2 (modulo minor differences in syntax). That is, all the *imap*s are evaluated strictly, and as a consequence, recursive array comprehensions fail with exception.

Secondly, we implement a flag `-force-letrec-imap` that implements the extensions from Section 3.1. That is, *imap*s that appear within letrecs like: `letrec x = imap` e_1 e_2 are put in closures and immediately forced, provided x occurs freely in e_2.

Both of the changes are implemented in a separate branch[4] called *force-imaps*. As a case study, we implement Cholesky decomposition in Heh: http://goo.gl/9a5dCA. When running the interpreter with the described flags, it can be seen that the program fails when all imaps are evaluated strictly and succeeds with the proposed approach. This demonstrates that the proposed approach is implementable, and indeed makes it possible to support recursive array comprehensions within strict array-based languages.

4.2 Compilation Approaches

After we have demonstrated how to extend an interpreter, we investigate how to implement a compiler for a language with recursive array comprehensions. At the time of writing, Heh supports compilation to SaC, but SaC does not support recursive array comprehensions. Therefore, we sketch a possible implementation in C. We focus on finding a runtime representation for values and on possible code generation for *imap*s and selections. We show

- that we can detect cycles in recursive imaps in sequential implementation; and
- how to parallelize recursive *imap*s using pthreads.

All the C code snippets below have been implemented. They can be seen as a core library used by code generators. When testing our implementation, we manually write programs in terms of the proposed library. For more details please refer to the *c-backend* directory in the Heh repository[5].

Core language. First, consider how the compiled code may look like for a simple version of the language without *imap*s with self-references. Let us start with data structures that describe a value:

```
// Kind of values
enum value_kind {
  k_scalar,
  k_evaluated_array,
  //...
};

// Scalar
struct scalar_value {
  enum value_kind kind;
  int val;
};

// Strict array
struct evaluated_array {
  enum value_kind kind;
  size_t size;
  int *val;
};

// A sum type for scalar and strict array.
typedef union value_union {
  enum value_kind vk;
  struct scalar_value vs;
  struct evaluated_array vev;
```

[2] In the context of this paper *shape* of array is its length.

[3] The implementation of Heh is freely available on GitHub: https://github.com/ashinkarov/heh

[4] Available at https://github.com/ashinkarov/heh/tree/force-imaps.

[5] See https://github.com/ashinkarov/heh/tree/force-imaps/c-backend.

```
  // ...
} value;
```

Evaluating *imap* and selections into a strict array can be defined as follows:

```
typedef value (*imap_fun) (value);

value eval_imap (size_t size, imap_fun f)
{
  struct evaluated_array v;

  v = mk_evaluated_array (size).vev;
  for (size_t i = 0; i < size; i++) {
    value vi = f (mk_scalar (i));
    v.val[i] = vi.vs.val;
  }

  return (value)v;
}

value eval_sel (value a, value idx)
{
  assert (a.val_kind == k_evaluated_array);
  assert (index_in_range (idx.vs.val,
                          a.vev.size));
  int el = a.vev.val[idx.vs.val];
  return mk_scalar (el);
}
```

Here `imap_fun` is a type for a function from index to scalar value. We omit definition of `mk_*` functions that create a fresh instance of the value of the chosen kind. We can see that *imap* updates a continuous piece of memory, and selections can access this memory without extra costs. In the general case, we have to perform a range check (`index_in_range` function) that is expressed as assertion, but very often this can be eliminated by static analysis.

Extended Language. Closures introduce a new `value_kind` name, a new function to construct a closure, and require an update of the selection function.

```
enum value_kind {
  // ...
  k_array_closure
};

struct array_closure {
  enum value_kind kind;
  size_t size;
  int *val;
  bool *mask;
  imap_fun f;
};

typedef union value_union {
  // ...
  struct array_closure vcl;
} value;
```

Selections and imaps are updated as follows:

```
value eval_imap_closure (size_t size, imap_fun f)
{
  return mk_array_closure (size, f);
}

value eval_sel (value a, value idx)
{
  // ... assert index in range ...
  switch (a.vk) {
  case k_evaluated_array:
    // This is the case from above
    return mk_scalar (a.vev.val[idx.vs.val]);

  case k_array_closure:
    if (a.vcl.mask[idx.vs.val])
      // If a value has been computed
      return mk_scalar (a.vcl.val[idx.vs.val]);
    else {
      // Compute array element at index idx
      value t = a.vcl.f (idx);
      a.vcl.mask[idx.vs.val] = true;
      a.vcl.val[idx.vs.val] = t.vs.val;
      return t;
    }
  }
}
```

As it can be seen, selections got more complex, and most importantly, every selection now has to check what kind of object we select from. This introduces a runtime overhead.

Note that as we require the first argument of an *imap* to be strict, we can immediately allocate memory for the final result. This means that selections into a closure never have to allocate additional memory, and when all the elements of the *imap* are computed, a closure can be turned into a strict array by swapping the pointer.

The implementation sketched so far implements the rules of the previous section. However, in doing so, it does not terminate in case the body of an *imap* contains cyclic dependencies in its element definitions. For example:

```
letrec x =
    imap 5 λi.if      i = 0 then a.1
              else if i = 1 then a.0
              else    1
```

The interesting observation here is that we can detect such a cycle easily almost at no cost. All it takes is a switch from a two-state boolean mask to a three-state mask, *e.g.* a mask of the following type:

```
enum val_status {
  s_no_value,
  s_in_progress,
  s_value
};
```

Then selections into a closure have to be adjusted as follows:

```
value
eval_sel (value a, value idx)
{
  // ...
  case k_array_closure:
    switch (a.vcl.mask[idx.vs.val]) {
    case s_value:
      return mk_scalar (a.vcl.val[idx.vs.val]);

    case s_no_value: {
      a.vcl.mask[idx.vs.val] = s_in_progress;
      value t = a.vcl.f (idx);
      a.vcl.mask[idx.vs.val] = s_value;
      a.vcl.val[idx.vs.val] = t.vs.val;
      return t;
    }

    default: // s_in_progress
      die ("cycle detected");
    }
  // ...
}
```

Finally, forcing imap closure in the letrec binding can be implemented as follows:

```
value
eval_imap_binding (var_t x, size_t size, imap_fun f)
{
  value v = mk_array_closure (size, f);
  bind_variable (x, v);
  // Force results
  for (size_t i = 0; i < size; i++) {
    switch (v.vcl.mask[idx.vs.val]) {
    case s_value:
      continue;
    case s_no_value:
      // trigger evaluation of (f idx)
      // store result in v
    default: //s_in_progress:
      die ("cycle detected");
    }
  }
  value r = mk_evakuated_array (size);
  r.vev.val = v.vcl.val;
  bind_variable (x, r);
  return r;
}
```

Note that in the implementation we can take a "shortcut" when it comes to implementing the Force rule. We do not have to call selection functions and accumulate their return values in a separate location. We can simply trigger function computations, for all the elements that are not yet computed and then swap result pointers.

4.3 Performance Considerations

4.3.1 Parallel execution. Currently, the evaluation of *imap*s forces an order in which array elements are evaluated. In the variant of our language with strict *imap*s, it can be shown that the order of evaluation can be arbitrary. This is because a functions are free of side-effects. As soon as we deal with *imap* closures this is no longer the case as selections into closures change the state of those closures, creating a side-effect. However, forcing all the elements of a closure can be done in arbitrary order.

When it comes to implementing the closure-free evaluation of *imap*s in parallel, we use a fork/join model. We cut the index-space of such an *imap* in n sub-spaces, spawn n threads, and wait until they finish. As an *imap* always evaluates to a single result, all the writes into memory are guaranteed to be disjoint. Therefore threads do not need to communicate and can write results into memory without using any locking.

In case of *imap* closures, parallel execution gets a bit more challenging. The index-space of an *imap* can be still divided into n subspaces, however, the property that each thread only writes the memory within the given subspace is lost. This happens because self-references may cause a cascade of writes at arbitrary indices. This means that threads have to alter the state of the mask and write results exclusively. Here is a sketch for the code for a selection operation that is capable to run in a multi-threaded context.

```
value eval_sel (value a, value idx)
{
  // ...
  case k_array_closure: {
    lock_mutex (mutex) {
      vs = a.vcl.mask[idx.vs.val];
      if (vs == s_no_value) {
        a.vcl.mask[idx.vs.val] = s_in_progress;
        compute_it = true;
      }
    }

    if (compute_it) {
      value t = a.vcl.f (idx);
      lock_mutex (mutex) {
        a.vcl.mask[idx.vs.val] = s_value;
        a.val_clarr.val[idx.vs.val] = t.vs.val;
      }
      return t;
    }

    if (vs == s_value)
      // return value
    else {
      // s_in_progress
      while (a.vcl.mask[idx.vs.val]
             == s_in_progress)
        ; // busy waiting

      return mk_scalar (a.vcl.val[idx.vs.val]);
    }
  }
  //...
}
```

We assume that `lock_mutex` is a macro that calls a function to lock the *mutex* at the beginning of the block and calls a function to unlock *mutex* at the end of the block.

The *mutex* variable is shared by all threads, and it is responsible to guarantee exclusive access to the mask and to the value of a closure. Each closure has to define its own *mutex*. In the code above we exclusively read the status of an element and check whether it has been computed. If it has not, we immediately change the status and set a local *compute_it* flag. We do the flip of the status immediately to avoid two threads computing the same value twice. Note that in principle, we can recompute results as often as we like, as we are in a functional setting. However, generally predicting when recomputing a dependency is faster than communicating with neighbouring threads is very difficult. Therefore we prefer communication over duplicated evaluation.

In case the value is marked as *s_in_progress* we cannot conclude that we found a cycle, as this status might have been set by a different thread. This means, that we have to wait for a status change for this value, which in turn means that the above code will not detect dependency cycles.

Regaining this ability in a multi-threaded execution is possible by for example maintaining a global graph of dependencies, checking for a cycle on every newly added dependency. This can be implemented in various ways, any such an implementation will create major overheads which are often impractical in high-performance setting.

4.3.2 Conditional within selections. Since our rules force the immediate evaluation of all *imap*s one might think that there is no need to keep a check within array selections whether the array is a closure or not. Unfortunately, we cannot easily statically decide whether a selection is potentially called while forcing a recursive *imap* or after the closure has been resolved. While it is possible to decide and optimise this in some cases, in general, this is undecidable. It might be possible to use a type system to further approximate

which selections can be specialised but further investigations into this issue lie outside of the scope of this paper.

4.3.3 Using Memoization. Another way to implement recursive *imap*s is to have it memoize its function, and force applications of first n elements. For example, the following recursive imap, assuming that n has been evaluated before:

```
letrec a = imap n λi.if i = 0 then 0
                     else a.(i-1)+1 in ...
```

can be treated as:

```
letrec a = λi.if i = 0 then 0 else a.(i-1)+1 in
letrec a' = memoize a in
letrec a = imap n λi.a' i in ...
```

In principle this works, however it requires a support for memoization. To make it efficient, a memoization mechanism should be instructed to keep memoized data in a continuous chunk of memory and forcing can be done without copying. In case memoization is not built-in, its implementation has to be able to express sharing of data with implicit updates efficiently. Usually one uses references, but they are not part of pure functional languages.

4.3.4 Evaluating Shapes Lazily. Our semantics insists that shapes of recursive *imap*s should be evaluated strictly. In principle, this requirement can be relaxed. For example one could envision an *imap* where its shape depends on the elements. However, the downside of such an approach, is that within the closure it is not possible to preallocate a continuous piece of memory for the result. This means that potentially, we introduce overheads when evaluating such a closure in parallel, as memory allocation would be a blocking operation, and it might be necessary to copy data from the closure to a strict array, in case the memory within the closure is not continuous.

4.4 Adoption into General Purpose Languages

Our approach to recursive array comprehensions is designed with array languages such as SaC in mind. An integration into those languages is rather straight-forward as outlined in the previous sections. This section explores the potential to transfer the key ideas into mainstream languages. Specifically, we look into what it would take to implement an *imap*-like construct in a strict language like OCaml and in a lazy language like Haskell. For OCaml we also show how our *imap* construct can be used in the context of general mutually-recursive *let*s to relax existing restrictions on the order of bindings, making the language to accept more programs.

4.4.1 In OCaml. Given the strict evaluation setting of OCaml, the built-in operator 'lazy' [21] makes it possible to convert any expression into a lazy value. The evaluation of such an expression is delayed, and can be triggered using the 'Lazy.force' operator. The 'lazy' operator changes the type of its argument from a to a *Lazy.t*.

In principle, such a lazy construct in a strict language is sufficient to implement what we propose: instead of an array of base type int, we can create an array of base type *int Lazy.t* and hide recursive selection self-references within the *imap* function behind the 'lazy' operator.

The downside of this solution is that we have to distinguish between strict and lazy values explicitly in the code. Therefore it gets harder to reuse functions in the body of an *imap*. Also, one would have to implement the forcing part explicitly for every recursive imap.

4.4.2 Improving Letrec in OCaml. Supporting recursive array comprehensions is very similar to dealing with a letrec with multiple variable bindings. Currently, in many strict languages like OCaml, in a letrec with multiple bindings, variables cannot directly depend on each other. For example, the following expression:

```
let rec a = 1 and b = 2 in a + b
```

is valid. However, an expression with mutual references to defined variables:

```
let rec a = b and b = 2 in a + b
```

is rejected with the error message that says: *This kind of expression is not allowed as right-hand side of 'let rec'*. However, an expression

```
let rec a = (fun x -> b) and b = 1 in a b
```

is perfectly valid. The problem here is that in order to satisfy letrec with mutual dependencies in a call-by-value language, one needs to guess the order in which expressions shall be evaluated. In general case this is undecidable statically. We can work around this problem by using the approach proposed in this paper. Assuming that *imap* as described in this paper would exist in OCaml, we could apply the following code transformation:

```
let rec                          let rec x = imap n (λi.
  x0 = e0                          if i = 0 then
  and                                e0[x.0/x0]...[x.(n-1)/xn-1]
  ...                              ...
  and                              else if i = n-1 then
  xn-1 = en-1                        en-1[x.0/x0]...[x.(n-1)/xn-1])
in e                             in
                                   let x0=x.0 and ... and xn-1=x.(n-1)
                                   in e
```

For a letrec with n variables x_0 to x_{n-1}, we create an array of n elements, where each element i evaluates an expression from the i-th binding, where all the accesses to variables x_0 to x_{n-1} are substituted with corresponding indexes into the newly created array. Further, in the goal expression of the letrec, we bind x_0 to x_{n-1} to the array elements $x.0$ to $x.(n-1)$.

The above expression demonstrates the basic idea. For full integration, we would have to support heterogeneous arrays, where all the elements could be of a different type. In OCaml terms — an n-element tuple. Then we would have to support *imap*-like constructs for such tuples, at least internally. Finally, we should be able to evaluate such a tuple lazily, which requires selections into a tuple to operate correctly on tuple-closures.

This idea is closely related to [15], where the authors propose a scheme that supports mutual recursion within a letrec with multiple bindings. However, as it seems, the order in which bound expressions are evaluated determines which values can be accessed within bound expressions safely. That is:

```
let rec a = 1 and b = a in ...
```

is allowed, but

```
let rec a = b and b = 1 in ...
```

is not supported.

4.4.3 In Haskell. definitions such as the ones above are valid in the first place:

```
let a = 2
    b = a in a + b
```

reduces to 4 as evaluation happens in a call-by-need manner. So the only consideration in this context is whether the explicit forcing of arrays would offer the same benefits in Haskell as it does in a strict setting such as SaC or OCaml. In the same way as in the strict setting, forcing *imap* closures in Haskell would avoid such closures to persist as well. However, as all the other evaluation happens under call-by-need semantics, the gain in loosing references to other arrays in the environment of *imap* closures has much less overall effect. References will persist in other closures that live in environments inhibiting update-in-place opportunities and potentially creating space leaks. Consider the following example:

```
let x = force (imap (10^5) λi.E[x]) in
let y = f x in ...
```

assuming that $E[x]$ is an expression that makes a reference to x. The $f\ x$ expression won't be evaluated up to the moment when y will be needed. Therefore, any operation on x like passing it as an argument, or referring to its elements, potentially requires forcing as well. Otherwise, our assumption about immediate memory reuse of arrays does not hold.

5 SPECULATIVE APPROACH

The syntactical restriction on recursive *imap*s that we introduce in Section 3.1 causes the evaluation of a number of programs to fail which would succeed under semantics where all *imap*s are evaluated lazily. For example, consider a program where the *imap* does not immediately appear on the right hand side of a letrec binding:

```
letrec x =
    if 1 then imap 10 λi.if i = 0 then 1 else x.0
    else 0
in ...
```

As *imap* does not appear syntactically immediately on the right hand side of the binding, the *imap* inside the conditional is evaluated strictly leading to a non-terminating execution.

We can improve our approach by extending the syntactic scope and check if the expression on the right hand side of a recursively defined variable contains an *imap* as a sub-expression which contains any free variable. However, such analysis quickly becomes tricky, as the *imap* could be placed inside a function body:

```
letrec f = λself.imap 10 λi.if i = 0 then 1
                                 else self.0 in
letrec c = f c in c
```

Another aspect of our approach that can be improved is the criterion which we use to decide whether to compute an *imap* lazily. Currently, we check whether the variable we bind is free. However, this check does not guarantee that this variable will ever be looked up during the evaluation. This means that we could have evaluated some of recursive *imap*s strictly, *e.g.*

```
letrec x = imap 10 λi.if 0 then x.0 else 1
```

which would allow us to further eliminate conditionals from the selection operations as discussed in Section 4.3.2.

In order to support all the programs that succeed under the semantics with lazy *imap*s, and still avoid creating and keeping closures unnecessarily, we can use a *speculative approach*. Informally, this can be summarised as follows: always start evaluating expressions strictly; in cases where we fail to look-up a variable, throw an exception. If an exception occurs in the process of letrec binding evaluation, restart the evaluation, but treat some of the *imap*s lazily.

We can fit this approach within the core semantics of our language. We do this by evaluating the right hand side of a letrec binding in a strict mode. In case a variable look-up hits the value $\perp$, we propagate $\perp$ up the call chain until it hits either an *imap* or a letrec that initiated the computation of the $\perp$ value. In case the value $\perp$ reaches an *imap*, the *imap* is turned into a closure. Finally, if the letrec binding evaluates to an *imap* closure, the closure has to be turned into a strict value.

There are several implications to using such an approach. First of all, letrecs will have to annotate $\perp$ values with a fresh identifier so that $\perp$ values do not escape the scope of a letrec evaluation. Consider an example:

```
imap 10 λi.letrec x = x in x
```

While evaluating letrec, a variable look-up will deliver $\perp$, but it should not turn the outer *imap* into a closure. This means that letrecs will have to start with $\perp_p$ and if the right hand side expression evaluates to $\perp_p$ then the evaluation should fail. Values $\perp_q$, where $p \neq q$ shall be propagated as the result of the definition.

When evaluating expressions of any other type (except letrec bindings or *imap*), $\perp_p$ values should be propagated as results, in case any of the dependent sub-expressions evaluate it. That is $\perp_p + 1 = \perp_p$, *etc.*

Finally, forcing should be able to bypass *imap* closures. Consider the following example:

```
letrec x = letrec y = imap 5 λi.if i = 0 then
                                    x.1 else 1
           in y
in x.0
```

A reference to x will turn the *imap* into a closure. The letrec that binds variable y will attempt to force this closure, but such a forcing will fail with $\perp_x$, as x is not yet bound. This means that if $\perp_p$ value appears during forcing, and $\perp_p$ was not initiated by the letrec we are currently evaluating, force has to return *imap* closure as a result. Also, to maintain termination properties, all such closures will have to be forced after the corresponding variable will be bound.

Despite the theoretical benefits that this approach could give, any practical implementation will come with very large overheads. Most importantly, evaluation of any expression will have to check whether any of the sub-expressions evaluate to $\perp$. Exception-handling techniques could help here, but for a realistic compiler implementation we find such an approach to be too expensive.

6 RELATED WORK

6.1 Laziness

Languages with support for lazy evaluation either do so for a special collection of data structures in a strict call-by-value setting, or they support full laziness. Lazy evaluation of data structures can

save memory use (space) and avoid performing unnecessary computation (time), *e.g.* when the size of a lazy data structure is pruned or when more information about an algorithm's solution is known during runtime *e.g.* [1]. Even if an entire lazy data structure is ultimately needed, only the parts required by a consumer/producer chain are manifested in memory at once [16].

Evaluation strategies are often defaults, and language primitives provide a switch to turn laziness on or off. For example, to inject laziness into the strict-by-default languages Scala [22] and Swift [17], the *lazy* keyword delays the creation of objects. Whereas to inject strictness, for example into Haskell code, bang patterns on data types force the manifestation of data structures into memory, and *seq* forces evaluations of functions, data types and bindings. These can be enabled for an entire module using the *StrictData* and *Strict* Haskell language extensions respectively.

Whilst lazy evaluation can yield time efficiencies by not carrying out unnecessary work, and space efficiencies by not manifesting entire data structure into memory at once, it can also have a desirable effect on the semantics of a program: turning a non-terminating program into a terminating one. For example, laziness can avoid evaluating $\perp$, exceptions, non-terminating computations and non-terminating evaluation of infinite data structures, if these values are never needed.

6.1.1 Lazy Producer/Consumer Patterns. The consumer/producer programming pattern is a technique for modularising programs into generators and selectors and is supported by lazy evaluation, such as the evaluation rules for *imap* in Section 3. A generator produces a potentially large number of solutions, and selectors choose the appropriate one.

A subset of the producer/consumer pattern is the ability to support lazy *sequences*, such as lists or streams, which is perhaps the most widely supported laziness feature for languages with laziness support. A user-defined Python generator [23] produces a sequence of results on-demand: the *yield* statement in a generator function computes the value to return with each *next ()* call by a consumer. Lazy sequences are supported in Clojure [9] with the *lazy-seq* macro, and many functions in Clojure's core library use it, such as the *repeat* and *iterate* generators, and list combinators including *map*, *filter* and *take*. Haskell's list combinators are also lazy, *e.g.* applying *head* to a list will preserve laziness in its tail. Scala's *Stream* data type shares the same laziness properties as Haskell lists and Clojure's seqable structures, whilst Scala lists are strict. Perl [26] also supports lazy lists with the '. . . ' operator and with the *gather/take* pattern. Haskell's support for lazy generators is more general than just sequences because all data types are lazy by default, *i.e.* Haskell generator functions can supply consumers with lazy lists, lazy trees, lazy arrays, lazy graphs, and so on.

6.1.2 Comparisons with imap. There are a number of key differences between languages with laziness support such as Scala, Swift, Perl, Python, Clojure and Haskell (Section 6.1.1), and the semantics of *imap*:

Random access Selection into sequences, *e.g.* as supported by Python, Scala, Haskell and Perl, is ordered: to access an element at position *n*, the sequence has to be consumed up to position *n*. In contrast, selections into an *imap* can be random. For example, selection of *A[4]* does not trigger evaluation of elements at positions *[0..3]* unless the computation at position *4* recurses into selections at these earlier positions.

Finite arrays Infinite sequence data structures are supported by some languages, including Scala *Streams*, Python generators, and lists in Clojure, Perl and Haskell. Our approach with *imap* is applicable in the context of finite arrays with a strict shape only.

Automatic laziness/strictness Unlike other languages with lazy evaluation support, including Scala, Swift and Haskell, in our approach there are no language annotations for switching between strict and lazy evaluation modes. Instead the evaluation semantics are built into the programming primitives: *imap* is lazy as shown in the Sel-Lazy-Imap-Full and Sel-Lazy-Imap-Empty selection rules in Section 3, and the rest of the language is strict (Section 2).

High performance + increased expressivity For high performance, array processing must have good CPU cache behaviour, such that array elements must be in cache at the time that they are to be processed. Lazy evaluation can result in poor cache performance, when too much time can be spent moving arrays between the heap in memory and the CPU for partial reduction. The programmer can overcome this using strictness annotations, or they can hope that strictness analysis *e.g.* [18] in compilers for lazy languages are successful enough in eliminating laziness in array processing pipelines. With a language design that combines strict array operators (Section 2) with the lazy *imap* (Section 3), the programmer gets the benefits of both: recursive array definitions with *imap* when this expressivity is needed, which memoises results for amortising computational complexity, within a strict array language that enables compiler optimisations [25] and good CPU cache performance.

6.2 I-Structures

6.2.1 Implicit Assignment with Laziness. The *imap* data structures is an extension to I-Structures [3] in Id [24], a functional language with non-strict semantics. I-Structures are allocated with the expression *I_array (m,n)*, which immediately returns an empty array, just as a lazy *imap* does. Assignment into the I-Structure and selection is explicit, with *A[i]=v* and *A[i]* respectively. The *imap* construct extends the I-Structure mechanism by making assignment implicit. It hides the explicit $A[i] =$ ‹*expr*› assignment operation on I-Structures, by instead forcing evaluation of $\lambda\ i.$ ‹*expr*› from the lazy generator.

6.2.2 Determinacy. The *imap* primitive shares a determinacy property with I-Structures, due to restrictions shared by both:

Write-once Elements are empty or full, and can only be written to once. Writes to each position in an *imap* are controlled by the Sel-Lazy-Imap-Empty rule, which writes the value of evaluating the $\lambda\ i.$ ‹*expr*› closure at each position. Subsequent selections use the Sel-Lazy-Imap-Full rule, and hence do not write to the *imap* again at this position.

Deferred reads If a selection is applied to an empty *imap* location, then there may be a delay until its value is computed. That is, a selection with the Sel-Lazy-Imap-Empty rule will

block on an empty element until its suspended $\lambda\ i.\ \langle expr \rangle$ expression is evaluated.

No test for emptiness As with I-Structures, there is no non-blocking test for emptiness at *imap* positions.

Thanks to this determinacy property, the timing of reads relative to writes do not change the final value of an *imap* because all selections at a position in an *imap* return a single, consistent value. Determinancy of reads/writes on I-Structures can be exploited for parallel execution on fine grained dataflow architectures [2], which can be supported for *imap*s too with careful implementation (Section 4.3.1).

6.3 Recursive Array Definitions

Only a few array-oriented languages make it possible to use self references in array comprehensions.

Accelerate [7] is an embedded array processing DSL inside Haskell with fully strict semantics, and targets multicore CPUs and GPUs. To avoid excessive SIMD divergence, it does not support recursive array definitions, contrasting with the expressive power of *imap* which allows recursive array definitions.

AlphaZ [29] is a generic set of tools for program analysis, transformation and parallelization in the Polyhedral Equational Model. An equational language named Alpha/Alphabets, which is a part of AlphaZ, makes it possible to specify a program as a set of dependencies between input and output data. Inputs and outputs are sets of points on an n-dimensional plane, where each set is bound by a polyhedron. Consider the LU decomposition problem[6]:

```
affine LUD {N|N>0}
input
  float A {i,j|1<=(i,j)<=N};
output
  float L {i,j|1<i<=N && 1<=j<i};
  float U {i,j|1<=j<=N && 1<=i<=j};
let
  U[i,j] = case
    {|1==i} : A[i,j];
    {|1<i} : A[i,j]
             - reduce(+, [k], L[i,k]*U[k,j]);
  esac;
  L = case
    {i,j|1==j} : A / (i,j->j,j)@U;
    {i,j|1<j} : (A - reduce(+, (i,j,k->i,j),
                            (i,j,k->i,k)@L
                            *(i,j,k->k,j)@U))
                / (i,j->j,j)@U;
  esac;
.
```

Input A is defined as a 2-dimensional rectangular array, indexed from 1 to N on every dimension. The outputs of this computation are sets L and U that have a triangular shape (lower/upper triangulars of $N \times N$ rectangular). The computation inside the let expression is a set of equations describing how elements of U depend on elements of A and L, and elements of L on elements of A and U. The let expression can define multiple bindings, where bound expressions can have mutual references.

Such a specification is passed to the polyhedral model [10] which tries to infer a schedule on how to traverse the points within the output sets, so that all the dependencies are respected. If such a schedule exists, the polyhedral tools are capable to generate a code for the given specification.

The main difference between AlphaZ and the approach proposed in this paper lies in the necessity to find a schedule statically. In case of *imap*, if a schedule is not found statically, the computation will be performed in a lazy fashion. In case of AlphaZ, if a schedule is not found, the specification is being rejected.

One of the extension proposals of APL [13] introduces the notion of recursive arrays. The proposal investigates how to deal with arrays of arbitrary nesting in the context of existing APL primitives. The proposed system makes it possible to treat trees of any shape as arrays, but neither the trees nor the specifications that construct them allow for self-references.

7 CONCLUSIONS

This paper presents a technique for adding recursive array comprehensions in a call-by-value array-oriented language. The key idea of the approach lies in treating arrays with possible self-references lazily, while the array is being computed. If a letrec binds an *imap* with potential self references, we create a closure for it, but we immediately force computations of all the elements within that closure, converting it into a strict array, if the *imap* is used.

This use of lazy evaluation yields three desirable properties for array programming:

Dependencies When implementing recursive array-based algorithms with existing strict array programming languages, the programmer must split recursive algorithms into non-recursive implementation phases, specifying recursive dependencies between them. With our approach, the programmer is alleviated from this requirement, since we support array expressions with self-references.

Expressive power With support for recursive array definitions, programs are shorter and are closer to their recursive mathematical definition. This is useful for real world applications, as shown in the two versions of the Choleski Decomposition in Section 1.

Parallelism Lazy *imap*s open up fine grained parallelism opportunities for compilers to exploit (Section 4.2). This makes it possible to avoid sequential bottlenecks in programs that have explicit dependency synchronisation points.

To ensure that programs benefit from the lazy *imap* evaluation rules without falling into the pitfalls of the performance overheads associated with laziness, lazy structures only reside in memory as long as they are needed. That is, *imap*s are fully evaluated in totality and hence can be garbage collected once used, or not at all. Dealing with strict values makes it possible to avoid overheads from running a conditional inside selection operations. This is important in the context of big data array processing.

7.1 Future Work

7.1.1 Exploiting Lazy imap. The *imap* evaluation rules in Section 3 could be exploited in several ways.

Dataflow analysis our criterion for evaluation of *imap*s lazily can be at times too restrictive. Currently, it is a syntactic restriction of the form `letrec x = imap`. This syntactic restriction can be eliminated by running a dataflow analysis

[6]This code is taken from: http://www.cs.colostate.edu/AlphaZ/wiki/doku.php?id=tutorial_lud

that checks whether a letrec variable can be found in any *imap* context.

Types for laziness Laziness and strictness properties for *imap*s could be lifted static type system rules, for exploitation during type-driven auto-parallelisation compilation.

Runtime laziness analysis Alternatively, runtime systems could exploit lazy *imap*s for parallelism, either tracking *imap* contexts or by preserving type information. However, runtime analysis with the speculative approach may come with substantial overheads (Section 5).

Eliminating laziness Some recursive *imap*s can be evaluated without creating a closure, given that we can choose an order of evaluation of elements that avoids forward dependencies. For example:

```
letrec a = imap 5 λi.if i = 0 then 0
                              else a.(i−1)+1  in a
```

can be compiled to the following C code:

```
int a[5];
for (size_t i = 0; i < 5; i++)
    a[i] = i == 0 ? 0 : a[i−1]+1;
```

given that we have fixed left-to-right order of index traversal.

Finding such an order of evaluation statically is a non-trivial task, and sometimes it will be undecidable (*e.g.* when an array index depends on data). Polyhedral model [10] in general, and AlphaZ [29] specifically can be leveraged here. Dependencies of an *imap* can be translated into a description of a polyhedron and then the model can attempt to find a schedule that respects these dependencies.

7.1.2 Machine Learning Guided imap Use. Machine learning approaches could be adopted for identifying performance-optimal use of the *imap* evaluation rules from Section 3, influenced by two recent approaches: (1) machine learning in [28] identifies optimal combinations of low level parallel implementations of strict high level functional array processing programs, and (2) machine learning in [8] identifies optimal combinations of adding laziness annotations into strict programs. An extension of our work could combine both, by profiling array programs to identify optimal combinations of injecting the lazy *imap* evaluation rules in real world array processing programs.

The approach presented in this paper can be generalised beyond array processing. The proposed technique can be used in other strict languages like OCaml (Section 4.4) to implement evaluation of recursive let expressions with arbitrary mutual recursion.

ACKNOWLEDGMENTS

This work was supported in part by grants EP/L00058X/1 and EP/N028201/1 from the Engineering and Physical Sciences Research Council (EPSRC).

REFERENCES

[1] Blair Archibald, Patrick Maier, Ciaran McCreesh, Robert Stewart, and Phil Trinder. 2018. Replicable Parallel Branch and Bound Search. *J. Parallel and Distrib. Comput.* 113 (March 2018), 92–114. https://doi.org/10.1016/j.jpdc.2017.10.010

[2] Arvind and David E. Culler. 1986. Annual Review of Computer Science Vol. 1, 1986. Chapter Dataflow Architectures, 225–253.

[3] Arvind, Rishiyur S. Nikhil, and Keshav Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 598–632.

[4] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 169–178. https://doi.org/10.1109/MEMCOD.2010.5558637

[5] Erik Barendsen and Sjaak Smetsers. 1993. *Conventional and uniqueness typing in graph rewrite systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 41–51. https://doi.org/10.1007/3-540-57529-4_42

[6] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. 2001. *Nepal — Nested Data Parallelism in Haskell.* Springer Berlin Heidelberg, Berlin, Heidelberg, 524–534. https://doi.org/10.1007/3-540-44681-8_76

[7] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011.* ACM, 3–14.

[8] Stephen Chang and Matthias Felleisen. 2014. Profiling for laziness. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* ACM, 349–360.

[9] Chas Emerick, Brian Carper, and Christophe Grand. 2012. *Clojure Programming - Practical LISP for the Java World.* O'Reilly.

[10] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (01 Feb 1991), 23–53. https://doi.org/10.1007/BF01407931

[11] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations* (4 ed.). John Hopkins University Press. http://www.ebook.de/de/product/20241149/gene_h_golub_matrix_computations.html

[12] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC - A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

[13] W. E. Gull and M. A. Jenkins. 1979. Recursive Data Structures in APL. *Commun. ACM* 22, 2 (Feb. 1979), 79–96. https://doi.org/10.1145/359060.359067

[14] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14).* ACM, New York, NY, USA, Article 88, 7 pages. https://doi.org/10.1145/2627373.2627388

[15] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. 2009. Compilation of Extended Recursion in Call-by-value Functional Languages. *Higher Order Symbol. Comput.* 22, 1 (March 2009), 3–66. https://doi.org/10.1007/s10990-009-9042-z

[16] John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107.

[17] Apple Inc. 2014. *The Swift Programming Language* (1st ed.). Apple Inc., USA.

[18] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. 1994. Efficient Strictness Analysis of Haskell. In *SAS*. 246–362.

[19] G. Kahn. 1987. Natural semantics. In *STACS 87*, FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 247. Springer Berlin Heidelberg, 22–39. https://doi.org/10.1007/BFb0039592

[20] John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (01 Dec 1995), 293–341. https://doi.org/10.1007/BF01018827

[21] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml.* O'Reilly Media, Inc, USA. http://www.ebook.de/de/product/19833727/jason_hickey_anil_madhavapeddy_yaron_minsky_real_world_ocaml.html

[22] Martin Odersky, Lex Spoon, and Bill Venners. 2011. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition* (2nd ed.). Artima Incorporation, USA.

[23] T. R. Padmanabhan. 2016. *Programming with Python.* Springer.

[24] Version Rishiyur, Rishiyur S. Nikhil, and Rishiyur S. Nikhil. 1991. ID Language Reference Manual. (1991).

[25] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[26] Randal L. Schwartz, Brian D. Foy, and Tom Phoenix. 2011. *Learning Perl - Making Easy Things Easy and Hard Things Possible: Covers Perl 5.14 (6th ed.).* O'Reilly.

[27] Artjoms Šinkarovs and Sven-Bodo Scholz. 2017. A Lambda Calculus for Transfinite Arrays: Unifying Arrays and Streams. *CoRR* abs/1710.03832 (2017). arXiv:1710.03832 http://arxiv.org/abs/1710.03832

[28] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* ACM, 205–217. https://doi.org/10.1145/2784731.2784754

[29] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. *AlphaZ: A System for Design Space Exploration in the Polyhedral Model.* Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-642-37658-0_2