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

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:

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

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:

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

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,