A Rosetta Stone for Array Languages

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Robert Bernecky
Snake Island Research Inc
Toronto, Ontario, Canada
bernecky@snakeisland.com

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, Scotland, UK
hv15@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

Abstract

This paper aims to foster cross-fertilisation between programming language and compiler research performed on different array programming language infrastructures. We study how to enable better comparability of concepts and techniques by looking into generic translations between array languages. Our approach is based on the idea of a basic core language HEH which only captures the absolute essentials of array languages: multi-dimensional arrays and shape-invariant operations on them. Subsequently, we investigate how to map these constructs into several existing languages: SAC, APL, JULIA, PYTHON, and C. This approach provides us with some first understanding on how the peculiarities of these languages affect their suitability for expressing the basic building-blocks of array languages. We show that the existing tool-chains by-and-large are very sensitive to the way code is specified. Furthermore, we expose several fundamental programming patterns where optimisations missing in one or the other tool chain inhibit fair comparisons and, with it, cross-fertilisation.

CCS Concepts • **Software and its engineering → General programming languages**; *Functional languages*;

Keywords functional languages, array languages, performance portability

ACM Reference Format:
Artjoms Šinkarovs, Robert Bernecky, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2018. A Rosetta Stone for Array Languages. In *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'18)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219753.3219754

ARRAY'18, June 19, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5852-1/18/06.
https://doi.org/10.1145/3219753.3219754

1 Introduction

Over time, several different languages with a focus on array processing have been defined and implemented. These reach from fairly long-standing languages such as APL or FORTRAN to more recent additions such as JULIA or FUTHARK. While all these languages differ in their syntax and the exact set of built-in operations, they do share the core setup: they support a data structure for representing multi-dimensional arrays, and they support a set of data-parallel map-/reduce-like array manipulation operations. Despite this similarity, cross-fertilisation between the languages and their implementations seems to be happening at a slow pace. Advanced implementation and optimisation techniques, runtime solutions and example codes developed for one language may never find their way into other array languages.

Transferring ideas from one setup to another often fails due to a perceived language barrier. This paper explores how to eliminate this barrier by looking into mechanical translations between different array languages while preserving the nature of array-based programs.

While the similarity in their core constructs should render a translation simple, in practice, it often turns out to be non-trivial: most languages offer several ways to translate the same program. These programs generate the same results, yet their runtimes may be orders of magnitude apart.

If we envision translation T as a function from any array language to any other array language:

$$T : \mathbb{X} \to \mathbb{X} \qquad \mathbb{X} = \{X_1, \ldots, X_n\}$$

it would be reasonable to avoid the need to create n^2 translators. We can do this by adopting L, an intermediate language for T:

$$T = LT \circ TL \qquad TL : \mathbb{X} \to \{L\} \qquad LT : \{L\} \to \mathbb{X}$$

T can be seen as a composition of the TL function that translates any language to L and the LT function that translates L to any other language. This reduces the number of translators from n^2 to $2n$, and given that L exists, can be seen as a universal representation of array-based programs.

L, itself, seems like a powerful idea, as it would simplify extensive benchmarking of array programs, and can be seen