Regular, Shape-polymorphic, Parallel Arrays in Haskell

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy[†]
Simon Peyton Jones[‡] Ben Lippmeier[†]

[†]Computer Science and Engineering, University of New South Wales
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Abstract

We present a novel approach to regular, multi-dimensional arrays in Haskell. The main highlights of our approach are that it (1) is purely functional, (2) supports reuse through shape polymorphism, (3) avoids unnecessary intermediate structures rather than relying on subsequent loop fusion, and (4) supports transparent parallelisation.

We show how to embed two forms of shape polymorphism into Haskell's type system using type classes and type families. In particular, we discuss the generalisation of regular array transformations to arrays of higher rank, and introduce a type-safe specification of array slices.

We discuss the runtime performance of our approach for three standard array algorithms. We achieve absolute performance comparable to handwritten C code. At the same time, our implementation scales well up to 8 processor cores.

Languages, Performance

1. Introduction

In purely functional form, array algorithms are often more elegant and easier to comprehend than their imperative, explicitly loop-based counterparts. The question is, can they also be efficient?

Experience with Clean, OCaml, and Haskell has shown that we can write efficient code if we sacrifice purity and use an *imperative array interface* based on reading and writing *individual array elements*, possibly wrapped in uniqueness types or monads Groningen [1997], Leroy et al. [2008], Launchbury and Peyton Jones [1994]. However, using impure features not only obscures clarity, but also forfeits the transparent exploitation of the data parallelism that is abundant in array algorithms.

In contrast, using a *purely-functional array interface* based on *collective operations* —such as maps, folds, and permutations— emphasises an algorithm's high-level structure and often has an obvious parallel implementation. This observation was the basis for previous work on algorithmic skeletons and the use of the *Bird-Meertens Formalism (BMF)* for parallel algorithm design Rabhi and Gorlatch [2003]. Our own work on *Data Parallel Haskell (DPH)* is based on the same premise, but aims at irregular data parallelism which comes with its own set of challenges Peyton Jones et al. [2008]. Other work on *byte arrays* Coutts et al. [2007b] also aims at high-performance, while abstracting over loop-based low-level code using a purely-functional combinator library.

We aim higher by supporting multi-dimensional arrays, more functionality, and transparent parallelism. We present a Haskell library of regular parallel arrays, which we call *Repa*[1] (Regular Parallel Arrays). While Repa makes use of the Glasgow Haskell Compiler's many existing extensions, it is a pure library: it does not require any language extensions that are specific to its implementation. The resulting code is not only as fast as when using an imperative array interface, it approaches the performance of handwritten C code, and exhibits good parallel scalability on the configurations that we benchmarked.

In addition to good performance, we achieve a high degree of reuse by supporting *shape polymorphism*. For example, `map` works over arrays of arbitrary rank, while `sum` decreases the rank of an arbitrary array by one – we give more details in Section 4. The value of shape polymorphism has been demonstrated by the language Single Assigment C, or SAC Scholz [2003]. Like us, SAC aims at purely functional high-performance arrays, but SAC is a specialised array language based on a purpose-built compiler. We show how to embed shape polymorphism into Haskell's type system.

The main contributions of the paper are the following:

- An API for purely-functional, collective operations over dense, rectangular, multi-dimensional arrays supporting shape polymorphism (Section 5).
- Support for various forms of constrained shape polymorphism in a Hindley-Milner type discipline with type classes and type families (Section 4).
- An aggressive loop fusion scheme based on a functional representation of delayed arrays (Section 6).
- A scheme to transparently parallelise array algorithms based on our API (Section 7)
- An evaluation of the sequential and parallel performance of our approach on the basis of widely used array algorithms (Section 8).

Before diving into the technical details of our contributions, the next section illustrates our approach to array programming by way of an example.

2. Our approach to array programming

A simple operation on two-dimensional matrices is transposition. With our library we express transposition in terms of a permutation operation that swaps the row and column indices of a matrix:

[1] Repa means "turnip" in Russian.