Data Parallel Haskell

Roman Leshchinskiy

Computer Science and Engineering
University of New South Wales
rl@cse.unsw.edu.au

With the advent of multicore CPUs, parallel programming is rapidly moving into the mainstream and expressive and efficient approaches to developing applications which fully utilise this hardware are urgently needed. Nested data parallelism (NDP) is an attractive model which allows complex parallel behaviour to be specified declaratively and liberates the programmer from low-level concerns such as synchronisation and communication. Collective operations on parallel arrays are the only means of expressing parallelism; but by allowing these to be arbitrarily nested, NDP transparently supports irregular computations and data structures. The main vehicle for compiling NDP programs is the *flattening transformation* which eliminates nested computations, producing code which contains only flat parallelism and can be efficiently executed on stock hardware.

The feasibility of the approach has been demonstrated by the programming language NESL which, however, had a severely restricted set of features and suffered from performance problems. The Data Parallel Haskell (DPH) project seeks to rectify these shortcomings by seamlessly integrating NDP into a well-known, feature complete functional language and by employing novel compilation techniques for generating efficient parallel code, with a particular focus on multi-core CPUs and shared-memory machines.

In this talk, we concentrate on the low-level aspects of implementing nested data parallelism in Haskell. The compilation of NDP programs involves transforming both the programmer-specified code and the representation of the data they operate upon. The resulting program must be extensively optimised if it is to exhibit competitive performance. Recent advances in type theory as well as the Glasgow Haskell Compiler's excellent optimisation capabilities allow us to implement many aspects of the compilation process and the underlying runtime system as a library without having to modify or extend the compiler itself.

NDP programs rely heavily on arrays with an inherently parallel semantics. Thus, it is not surprising that most algorithms are expressed as pipelines of array computations. A naive implementation which executes each computation separately, producing a large number of intermediate arrays, has unacceptable performance. To generate efficient code, the intermediate arrays must be eliminated by *fusing* array producers with consumers. This problem has been studied extensively but only for sequential programs. Fortunately, fusion of *parallel* array computations can be formulated as a two-phase process by separating synchronisation points from purely parallel code. Removing superfluous synchronisations allows us to subsequently fuse the latter using conventional techniques. By expressing the separation on the type level we are able to express the entire fusion process as a set of algebraic rewrite rules which are directly supported by the Glasgow Haskell Compiler.