Development of Parallel Algorithms in Data Field Haskell[*]

Jonas Holmerin[1] and Björn Lisper[2]

[1] Department of Numerical Analysis and Computing Science, Royal Institute of Technology, S-100 44 Stockholm, SWEDEN, joho@nada.kth.se
[2] Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, SWEDEN, bjorn.lisper@mdh.se

Abstract. Data fields provide a flexible and highly general model for indexed collections of data. Data Field Haskell is a dialect of the functional language Haskell which provides an instance of data fields. We describe Data Field Haskell and exemplify how it can be used in the early phase of parallel program design.

1 Introduction

Many computing applications require indexed data structures. The canonical indexed data structure is the array. However, for sparse, distributed applications, other, more dynamic indexed data structures are needed. It is desirable to develop such algorithms on a high level first, in order to get them right, since the low level data representations can be intricate.

Data Field Haskell provides an instance of data fields – a data type for general indexed structures. This Haskell dialect can be used for rapid prototyping of parallel computational algorithms which may involve sparse structures.

Various versions of the data field model have been described elsewere [1–4]. The contribution of this paper is a description of an implementation and an example of how it can be used in parallel program design.

2 The Data Field Model

Data fields are based on the more abstract model of indexed data structures as functions with finite domain [1, 2]. This model is simple and powerful, but for real implementations explicit information about the function domains is needed. Data fields are thus entities (f, b) where f is a function and and b is a *bound*, a set representation which provides an upper approximation of the domain of the corresponding function. We require that the following operations are defined for bounds:

– An interpretation of every bound as a set.

[*] This work was supported by The Swedish Research Council for Engineering Sciences (TFR), grant no. 98-653.

– A predicate classifying each bound as either *finite* or *infinite*, depending on whether its set is surely finite or possibly infinite.
– For every bound b defining a finite set, $size(b)$ which yields the size of the set and $enum(b)$ which is a function enumerating its elements.
– Binary operations $\sqcap$, $\sqcup$ on bounds ("intersection", "union").
– The bounds *all* and *nothing* representing the universal and empty set, respectively.

These operations are chosen to support the usually assumed set of collection-oriented operations [7] without revealing the inner structure of the bounds. An important derived operation is *explicit restriction*: $(f, b) \downarrow b' = (f, b' \sqcap b)$. The theory of data fields also defines *φ-abstraction*, a syntax for convenient definition of data fields which parallels λ-abstraction for functions. See [3, 4].

3 Data Field Haskell

`datafield`	defines data field from function and bound
`!`	data field indexing
`bounds`	the bound of a data field
`outofBounds`	an out-of-bounds error value
`predicate`	forms predicate bound from predicate
`join`, `meet`	"union" and "intersection" of bounds
`<\>`	explicit restriction of data field with bound
`inBounds`	checks if an element belongs to the set defined by a bound
`foldlDf`	folds (reduces) finite data field w.r.t. binary operation

Table 1. Some operations on data fields and bounds.

Data Field Haskell is a Haskell dialect where the arrays have been replaced by an instance of data fields, a variation of the *sparse/dense arrays* of [3, 4]. Our implementation of Data Field Haskell is based on the NHC compiler [6] for Haskell v. 1.3. The implementation is sequential and we have not implemented any advanced optimizations.

Data Field Haskell has data types `Datafield a b` for datafields and `Bounds a` for the corresponding bounds. Table 1 lists some functions for data fields and bounds. It has a rich variety of finite and infinite bounds: *dense bounds*, i.e., traditional array bounds, *sparse finite bounds*, which represent general finite sets, *predicate bounds*, which are classified as infinite, `universe` which represents the universal set, and `empty` which represents the empty set. *Product bounds* represent Cartesian products and generalise multidimensional array bounds.

3.1 Forall- and For-abstraction

Data Field Haskell provides φ-abstraction, with syntax similar to λ-abstraction in Haskell [5]:

`forall` *apat₁* ... *apatₙ* `->` *exp*

The semantics of `forall x -> t` is `datafield (\x -> t) b`, where `b` is computed from bounds of data fields in `t` as to give an upper approximation of the domain of `\x -> t`. It can be thought of as an implicitly parallel, functional `forall` statement where first `b` is computed and then, if needed, `\x -> t` is computed for all `x` in `b`.

`for`-abstraction provides a convenient syntax to define a data field by cases over different parts of its domain. The syntax is

`for` *pat* `in {` e_1 `->` e'_1 `;` ... `;` e_n `->` e'_n `}`

where the e_i are bounds and the e'_i are data field values. The semantics is a data field whose bound is restricted by the union of $e_1, \ldots, e_n$ and whose value for each `x` is e'_i `! x`, for the lowest i such that `x` belongs to e_i.

4 An Example

We exemplify the use of Data Field Haskell with simulation of a system of particles over a range of time. Each particle i has a state $(\bar{r}_i, \bar{v}_i)$ with a position $\bar{r}_i$ and a velocity $\bar{v}_i$. The state transition function (1) gives the new state after time Δt. These equations are parameterized w.r.t. f_i, which yields the acceleration a_i for each particle i from the spatial distribution of the particles. They are more or less directly expressed[1] in Data Field Haskell in Fig. 1. The result is an executable, dimension-polymorphic specification of particle simulation with time step `dt`.

$$\begin{aligned} (\bar{r}_i, \bar{v}_i) &\mapsto (\bar{r}_i + \bar{v}_i \cdot \Delta t, \bar{v}_i + \bar{a}_i \cdot \Delta t), \quad i \in \mathit{Particles} \\ \bar{a}_i &= f_i(\langle\, r_j \mid j \in \mathit{Particles}\,\rangle), \quad i \in \mathit{Particles} \end{aligned} \tag{1}$$

This specification can be manually refined into a more explicitly parallel algorithm for particle simulation by *distributing* the particle state onto a set of processors, and *localizing* the parallel algorithm by neglecting long-range interactions. The resulting parallel algorithm has a distributed state which consists of a predicate, a sparse particle datafield, and a *neighbourhood bound* for each processor. A processor "owns" each particle in the area defined by its predicate. After each iteration every processor tests which particles it now owns. This test is localized to particles originating from the neighbours only. This approximation is correct only if long-range interactions can be ignored, for instance if we

[1] The explicit restriction in `p_newstate` is necessitated by a flaw in the current derivation of bounds for `forall`-abstraction. We expect to rectify it in later releases of Data Field Haskell.

```
data Pstate = PS (Datafield Int Float) (Datafield Int Float)
pos (PS r _) = r
vel (PS _ v) = v

p_newstate dt f s =
  forall i ->
    let ai = f i s in
      PS ((forall k-> ((pos (s!i))!k) + ((vel (s!i))!k)*dt)
            <\> bounds (pos (s!i)))
         (forall k-> ((vel (s!i))!k) + (ai!k)*dt)


data Dstate proc_id part_id = DS ((Datafield Int Float) -> Bool)
                                 (Datafield part_id Pstate)
                                 (Bounds proc_id)
pred  (DS p _ _) = p
state (DS _ s _) = s
neigh (DS _ _ b) = b

prunion d1 d2 =
  forall x -> if (inBounds x (bounds d1)) then d1!x else d2!x
prUnion = foldlDf prunion (datafield (\x -> outofBounds) empty)

d_newstate dt f dstate =
  forall p ->
    DS (pred (dstate!p))
       (let {
          neighstate = prUnion
                       (for pn in neigh (dstate!p) -> state (dstate!pn));
          newstate   = (p_newstate dt f neighstate) }
        in newstate <\> predicate (\j -> pred (dstate!p) (pos (newstate!j))))
       (neigh (dstate!p))
```

Fig. 1. Data Field Haskell solutions for the simulation problem: executable specification, and parallelised version.

simulate molecules which interact through collisions only. The set of neighbour processors is described, for each processor, by its neighbourhood bound. Fig. 1 also gives the data type and transition function `d_newstate` for the distributed particle simulation. `d_newstate` first computes, for each processor, the union of the particles over the neighbours, then a local state transition for these, and finally it "masks out" the particles which now belong to it. (Since data fields are not sets we must instead of set union use "`prunion`", which gives priority to its first argument for indices where both arguments are defined.) `d_newstate` does not modify the predicate field, but this is possible and could be used to model adaptive methods where areas for different processors are changed to balance the load.

5 Conclusions

We have presented Data Field Haskell, a Haskell dialect with data fields which supports a highly flexible form of collection-oriented programming. A possible application is for rapid prototyping in the early specification phase of parallel algorithms. We exemplified with the specification and initial development of a simple parallel particle simulation algorithm. The resulting algorithm specification is dimension-polymorphic and easily adaptable to regular and irregular grids, static or dynamic mappings, and different target architectures.

References

1. P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 210–222. ACM Press, June 1993.
2. B. Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Vol. 1132 of *Lecture Notes in Comput. Sci.*, pages 220–251, Les Ménuires, France, Mar. 1996. Springer-Verlag.
3. B. Lisper. Data fields. In *Proc. Workshop on Generic Programming*, Marstrand, Sweden, June 1998. `http://wsinwp01.win.tue.nl:1234/WGPProceedings/`.
4. B. Lisper and P. Hammarlund. The data field model. Technical Report TRITA-IT R 99:02, Dept. of Teleinformatics, KTH, Stockholm, Mar. 1999. `ftp://ftp.it.kth.se/Reports/TELEINFORMATICS/TRITA-IT-9902.ps.gz`.
5. J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. L. Peyton Jones, A. Reid, and P. Wadler. Report on the programming language Haskell: A non-strict purely functional language, version 1.4, Apr. 1997. `http://www.haskell.org/definition/`.
6. N. Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1995.
7. J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, Apr. 1991.