DEVELOPMENT AND VERIFICATION OF PARALLEL ALGORITHMS IN THE DATA FIELD MODEL*

BJÖRN LISPER† AND JONAS HOLMERIN‡

Abstract. Data fields are partial functions provided with explicit domain information. They provide a very general, formal model for collections of data. Algorithms computing data collections can be described in this formalism at various levels of abstraction: in particular, explicit data distributions are easy to model. Parallel versions of algorithms can then be formally verified against algorithm specifications in the model. Functions computing data fields can be directly programmed in the language Data Field Haskell. In this paper we give a brief introduction to the data field model. We then describe Data Field Haskell and make a small case study of how an algorithm and a parallel version of it both can be specified in the language. We then verify the correctness of the parallel version in the data field model.

1. Introduction. Many computing applications require indexed data structures. In many applications the indexing capability provides an important part of the model. On the other hand, the memory space of a parallel computer architecture is also indexed. Thus, indexed structures can describe data close to both the problem domain and architectural domain.

The canonical indexed data structure is the array. However, in particular when dealing with sparse, distributed applications, other, more dynamic indexed data structures are needed. Their low-level representations can be intricate. Thus, it is hard to design algorithms using them from scratch, in particular since the algorithms themselves may be complex, and to port algorithms to different architectures. Therefore, it is desirable to develop such algorithms on a high level first, where implementation details are hidden.

The data field model[1] is a formal model where indexed data structures are modeled as partial functions supplied with explicit information about their domains. A small formal language can be used to define data fields mathematically, and known proof techniques can be used to prove properties about data fields defined within the language. In particular, it is interesting to prove correspondences between different data fields, since this can be used to prove the correctness of refinement steps taken in a design process where abstract specifications are successively refined into distributed implementations.

However, it is also helpful to validate such steps experimentally, in particular since they sometimes are only approximately correct and can yield, for instance, different numerical properties. The language Data Field Haskell can be used for this. It provides an instance of data fields, which is general and flexible enough to specify data fields both on a very high level of abstraction and on a level with explicitly parallel data distributions.

The rest of this paper is organized as follows. Sect. 2 gives a brief description of the underlying data field model. Sect. 3 describes Data Field Haskell. Sect. 4 provides

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

†Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, SWEDEN, bjorn.lisper@mdh.se

‡Department of Numerical Analysis and Computing Science, Royal Institute of Technology, SE-100 44 Stockholm, SWEDEN, joho@nada.kth.se

[1] "Field" should be understood as in physics, as an entity that is a function of space and possibly time.

an example of how a parallel program can be developed from a specification in Data Field Haskell. In Sect. 5 we verify formally that the parallel program indeed is correct w.r.t. the original specification. Sect. 6 provides an account for related work. In Sect. 7, finally, the story is wrapped up. The limited space does not allow a complete description of Data Field Haskell here – see [10] for the details.

Various versions of the data field model have been described elsewhere [7, 12, 13, 14]. The contribution of this paper is a description of a concrete implementation, an example of how it can be used in parallel program design, and a demonstration how the model can be used for formal verification of parallel programs.

2. The Data Field Model.

2.1. Partial Functions. The concept of data fields is based on the more abstract model of indexed data structures as functions with finite domain [7, 12]. An array with range `[1..n]`, for instance, can be seen as a function from $\{1, \ldots, n\}$, but we could also model "irregular" indexed structures as functions with non-contiguous, possibly non-numerical domains. Calls to a function outside its domain return an error value "$*$", with algebraic properties similar to the divergent element $\perp$.

We define partial functions in a variation of the metalanguage for continuous functions in [23], extended with the constant $*$. Within this language, we can now define most types of collection-oriented operations [20] as higher order functions operating on partial functions [7, 14]. The language, however, lacks operations that extract the domain of a function or any information pertaining to it, like the size ("information operations" as defined in [20]). The lack of domain information makes it impossible to define collective operations such as *reduction*, since they need to know which elements to include. A partial function f over an enumerable cpo models a data structure that contains elements $f(x)$ exactly for those fully defined[2] x where $f(x) \neq *$. We thus define, for all partial functions f, its *domain* $dom(f) = \{\, x \mid f(x) \neq * \text{ and } x \text{ fully defined} \,\}$. $dom(f)$ is not computable in general, but we can still use it in abstract algorithm specifications.

Now assume, for any enumerable cpo C under consideration and any finite $C' \subseteq C$, with fully defined elements only, a bijection $i_{C'}\colon \{0, \ldots, |C'|-1\} \to C'$. Equipped with these functions, and the set operation $|\cdot|$ (cardinality), we define:

$$\begin{aligned} size(f) &= |dom(f)| \\ enum_f &= i_{dom(f)} \end{aligned}$$

We can now define reduction over nonempty, partial functions with finite *size* w.r.t. a binary operation $\oplus$:

$$red(\oplus, f) = red'(\oplus, f, size(f) - 1)$$

where

$$\begin{aligned} red'(\oplus, f, 0) &= f(enum_f(0)) \\ red'(\oplus, f, n+1) &= red'(\oplus, f, n) \oplus f(enum_f(n+1)) \end{aligned}$$

If $\oplus$ has an identity element e, then we define $red(\oplus, \lambda x.*) = e$ for the empty partial function $\lambda x.*$. Note the analogy with reduction over lists in the Bird-Meertens

[2] x is fully defined, or *maximal*, in the cpo $\langle C, \sqsubseteq \rangle$ if $\forall y. x \sqsubseteq y \implies x = y$. Unless the data structure has a lazy lookup function, a lookup is successful only if the index is fully defined.

formalism [1]: we indeed have

$$red(\oplus, f) = \oplus/[f(enum_f(0)), \ldots, f(enum_f(size(f) - 1))]$$

We will develop this analogy further in Sect. 5, where we will use the partial function model to verify the correctness of a parallelised algorithm.

2.2. Data Fields. The partial function model is simple and powerful, but in order to actually implement it explicit information about the function domains is needed. Thus we consider entities (f, b) – the data fields – where f is a function and b is a *bound*, a set representation that bounds the domain of the corresponding function. We require that the following operations are defined for bounds:

- For each bound b an interpretation $[\![b]\!]$ as a predicate (which in turn defines a set $\{\{b\}\} = \{\, x \mid [\![b]\!](x) = true \text{ and } x \text{ fully defined} \,\}$).
- 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, $size(b)$ that yields the size of the set and $enum(b)$ that is a function enumerating its elements.
- Binary operations $\sqcap$, $\sqcup$ on bounds ("intersection", "union").
- Bounds representing the universal and empty set, respectively.

These operations are chosen to support the operations on partial functions that require the domain of the functions. They must fulfil certain properties, see [14]. We define *data field application* ("lookup") viz.:

$$(f, b) \,!\, x = if([\![b]\!](x), f(x), *) \quad x \text{ fully defined}$$
$$(f, b) \,!\, x = \bot \quad \text{oherwise}$$

Every data field $d = (f, b)$, where $f\colon \alpha \to \beta$, then defines a partial function $[\![d]\!]_{\alpha\to\beta}$ through $[\![d]\!]_{\alpha\to\beta}(x) = d \,!\, x$ for all x. It immediately follows that $dom([\![(f, b)]\!]_{\alpha\to\beta}) \subseteq \{\{b\}\}$. Thus, the bound of a data field always bounds the domain of its function. The bound does not have to be tight – we can have $(f, b) \,!\, x = *$ even for some $x \in \{\{b\}\}$.

The theory of data fields also defines *φ-abstraction*, a syntax for convenient definition of data fields that parallels λ-abstraction for functions. The idea is that $[\![\varphi x.t]\!]_{\alpha\to\beta}$ should be the same function as $\lambda x.t$, except possibly for some pathological cases. Different definitions of $\varphi x.t$ fulfilling this are possible: some are given in [14]. For these definitions it is possible to prove the following result, which is stated more precisely in [14]:

THEOREM 2.1. *If the bound of $\varphi x.t$ is fully defined, then $\varphi x.t \,!\, y = \lambda x.t\, y$ for all fully defined y.*

Theorem 2.1 holds under the condition that $\bot$ and $*$ are identified. Even if not, it still holds for total functions (i.e., which return a fully defined result given fully defined arguments). For data fields that define total functions, the result thus means that formal reasoning about data fields can be carried out in the more abstract model of partial functions. We will use this in Sect. 5.

3. Data Field Haskell. 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 [13, 14]. Our implementation of Data Field Haskell is based on the NHC compiler [18] for Haskell v. 1.3. The implementation is sequential and we have not implemented any advanced optimizations. In the following the reader is assumed to have a working knowledge of Haskell.

`datafield`	defines data field from function and bound
`!`	data field indexing
`<:>`	forms dense bound from pair of index tuples
`sparse`	forms finite sparse bound from list
`predicate`	forms predicate bound from predicate
`<*>`	forms product bound from two bounds
`prod_`n	forms n-dimensional product bound
`inBounds`	checks if an element belongs to the set defined by a bound
`finite`	tests bound for finiteness
`size`	the number of elements in a finite bound
`enumerate`	list of elements of the set defined by a finite bound in given order
`join`, `meet`	$\sqcup$ and $\sqcap$ on bounds
`lowerBound`	first enumerated element in finite bound
`upperBound`	last enumerated element in finite bound
`<\>`	explicit restriction of data field with bound
`foldlDf`	folds (reduces) finite data field w.r.t. binary operation
`hstrictTab`	evaluates all elements in finite data field fully
`outofBounds`	the out-of-bounds error value $*$
`isoutofBounds`	test for `outofBounds`

```
(bx1 <*> by1) `op` (bx2 <*> by2)  =  (bx1 `op` bx2) <*> (by1 `op` by2)
            universe `meet` x     =  x `meet` universe = x
               empty `join` x     =  x `join` empty = x
            universe `join` x     =  universe
            x `join` universe     =  universe, x fully defined
               empty `meet` x     =  empty
               x `meet` empty     =  empty, x fully defined
      bounds (datafield f b)      =  b
     (datafield f b1) <\> b2      =  datafield f (b2 `meet` b1)
```

`meet`	E	U	S	D	P	$\times$
E	E	E	E	E	E	E
U		U	S	D	P	$\times$
S			S	S	S	S
D				D	S	$\times$
P					P	P
$\times$						$\times$

`join`	E	U	S	D	P	$\times$
E	E	U	S	D	P	$\times$
U		U	U	U	U	U
S			S	S	P	S/P
D				D	P	$\times$
P					P	P
$\times$						$\times$

TABLE 3.1

Selected operations on data fields and bounds, some algebraic laws, and tables for result "types" of `join` *and* `meet` *as a function of the argument "types". "*`op`*" is any of* `join`, `meet`. $E =$ `empty`, $U =$ `universe`, $S =$ `sparse`, $D =$ `dense`, $P =$ `predicate`, $\times =$ *product bound. "S/P" in the table for* `join` *means that the result is* `sparse` *if the product bound is finite, and a* `predicate` *otherwise. (Backquotes around binary functions, like in* `` `meet` ``*, turn them into infix operators in Haskell.)*

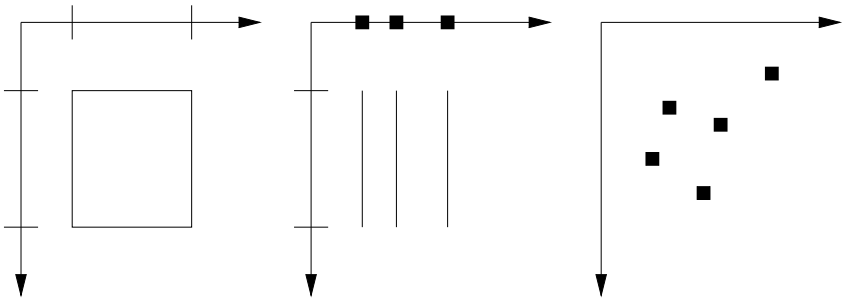

FIG. 3.1. *Some two-dimensional bounds: two product bounds, and a sparse two-dimensional bound.*

Data Field Haskell has data types `Datafield a b` for datafields and `Bounds a` for the corresponding bounds. There are basic functions to form data fields and different kinds of bounds, and the required operations on bounds in Sect. 2.2 are available. There are also some useful derived operations. Table 3.1 lists the most important functions, and gives some algebraic laws for them. It also lists the kind of bound computed by `join` and `meet` for different kinds of argument bounds.

Data Field Haskell 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. Some two-dimensional bounds are illustrated in Fig. 3.1.

3.1. Forall-abstraction. Data Field Haskell provides φ-abstraction, with the following syntax (described in the metasyntax of the Haskell report [17]):

`forall` $apat_1$ ... $apat_n$ `->` exp

Thus, the syntax is analogous to λ-abstraction in Haskell. The semantics is, with some minor deviations, the same as for φ-abstraction in [14]. `forall x -> t` can be thought of as an implicitly parallel, functional `forall` statement where first its bound `b` is computed and then, if needed, `\x -> t` is computed for all `x` in `b` in any (possibly parallel) order. The rules for computing `b` generalise existing array language principles for computing implicitly given bounds in the following cases: elementwise applied operations, selection of row/column from multidimensional array, and translation with constant offset. The exact rules are given in [10]: some representative examples are shown in Table 3.2. (The reader is encouraged to compare the bounds of the `forall`-abstractions with the domains of the partial functions defined by the corresponding λ-abstractions.)

4. An Example. We now give a simple example how a prototype for parallel code, which could be taken as a starting point for a real implementation, can be stepwise derived in Data Field Haskell from a specification. Our example is Jacobi's algorithm, which is a classical, iterative method to solve linear systems of equations. Iterative methods of this kind are interesting to apply in particular to very large sparse systems, where direct methods can be prohibitively expensive. Thus, interesting issues in the development of parallel algorithms for Jacobi's metod are the ability to handle

```
                          bounds (forall x -> 17)  =  universe
                  bounds (forall x -> outofBounds)  =  empty
                    bounds (forall x -> a!x + b!x)  =
                      (bounds a) `meet` (bounds b)
                bounds (forall (x,y) -> a!x + b!y)  =
                         (bounds a) <*> (bounds b)
     bounds (forall x -> if a!x then b!x else c!x)  =
  (bounds a) `meet` ((bounds b) `join` (bounds c))
                       bounds (forall x -> a!(1,x))  =  b2
                       bounds (forall x -> a!(x,x))  =  b1 `meet` b2
                  bounds (forall (x,y) -> a!(y,x))  =  b2 <*> b1
bounds (forall x -> (datafield f (1 <:> 5))!(x+1))  =  0 <:> 4
```

TABLE 3.2
Some examples of bounds for `forall`*-abstraction. Here,* `bounds a = b1 <*> b2`.

```
sumDf = foldlDf (+) 0

iter f x conv =
  let xnew = f x
  in if (conv xnew x) then xnew else iter f xnew conv

jacobi a b eps x = iter (jacobi_iter a b) x (conv eps)

jacobi_iter a b x =
  forall i -> ((b!i) - sumDf ((forall j -> a!(i,j)*(x!j))
                              <\> predicate (\j -> j /= i)))/a!(i,i)

conv eps x y = maxnorm (forall i -> (x!i - y!i)) < eps

maxnorm x = foldl1Df max (forall i -> abs (x!i))
```

FIG. 4.1. *Data Field Haskell specification of Jacobi's algorithm.*

sparse structures at a suitable level of abstraction, and also how to develop methods for load balancing that are both reasonably efficient and give a reasonably good result. Notably, it should be possible to easily plug in different methods for load balancing since it is common to have matrices with a certain structure, for which specialised methods for load balancing may exist.

Jacobi's method solves the equation $Ax = b$, where A is an $n \times n$-matrix and x, b are n-vectors, by computing iterates $x^{(k)}$ according to

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}}{a_{ii}}, \quad i = 1, \ldots, n \tag{4.1}$$

until two successive iterates satisfy some convergence criterion, for instance that $\|x^{(k+1)} - x^{(k)}\| < \epsilon$. Here, $\|\cdot\|$ is a vector norm: in this example, we use the maximum norm $\|y\| = \max_{i=1}^{n} |y_i|$.

A specification in Data Field Haskell of Jacobi's algorithm, according to (4.1), is given in Fig. 4.1. Here, `sumDf` sums all the elements in a finite data field (`foldlDf`

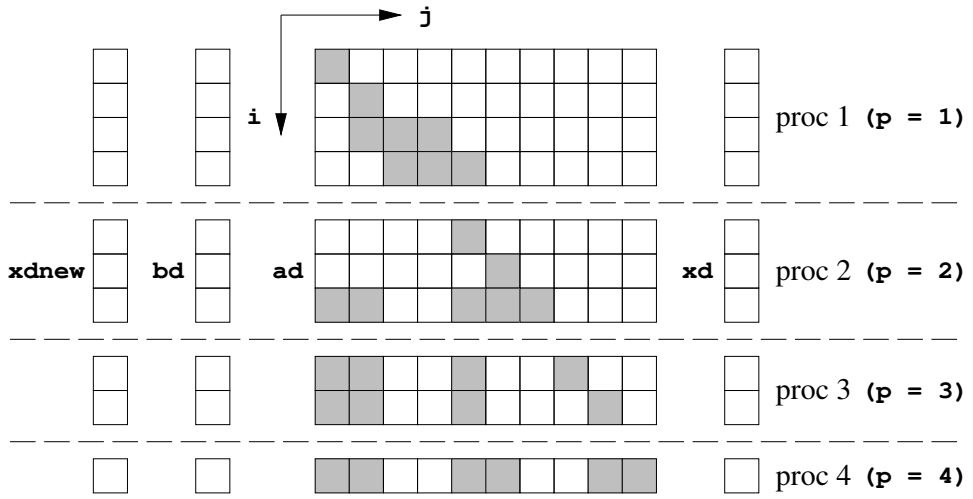

FIG. 4.2. *Distribution of data in the parallel Jacobi algorithm.*

is analogous to the Haskell function `foldl` on lists), and `iter` is a tail-recursive skeleton for general iterative algorithms with a convergence check. `jacobi` iterates `jacobi_iter` repeatedly until convergence according to `conv` is reached. Note the close resemblance of the definition of `jacobi_iter` and (4.1) – this exemplifies the ability to program with `forall`-abstraction in a style very close to mathematical notation. `conv`, finally, uses `foldl1Df`, which is analogous to `foldl1` over lists.

Note that the bounds of `a` do not show up explicitly in the code: especially, the sums over `forall j -> a!(i,j)*(x!j)` have their limits given implicitly by the bounds of this data field expression, for the different values of `i`. These bounds are the `meet` of the bounds for `forall j -> a!(i,j)` and `x`. If `bounds a` equals `(1,1)<:>(n,n)` (representing a dense $n \times n$-matrix) and if `bounds x = 1<:>n` (a dense n-vector) then the `meet` is `1<:>n`, and the sum will thus be performed over all `j` $\in \{1, \ldots, n\}$. If, however, `a` has a sparse bound, then `forall j -> a!(i,j)` will have a sparse bound as well[3] and the `meet` will also be sparse: thus, only the defined elements will be summed. The algorithm specification in Fig. 4.1 therefore defines both a dense and a sparse algorithm depending on the kind of data field!

We now manually refine this specification into a parallel algorithm. The idea is to distribute the rows of `a` and the elements of `b` according to a partitioning of the interval $\{1, \ldots, n\}$ as indicated in Fig. 4.2. The inner products can then be kept local, which keeps the communication down, and this design decision leads to a parallel algorithm where each processor computes a segment of `xnew`. These are then assembled and broadcast for the new parallel iteration. The convergence test can be performed as a parallel reduction. If `a` is sparse, then it can pay off to perform a nontrivial load balancing, which however can be made statically since the structure of `a` does not change during the course of the algorithm. Our algorithm takes a static load balancing as a parameter.

The distributed program is shown in Fig. 4.3. It makes heavy use of nested data fields, where the outermost data fields have indices (`p` in Fig. 4.3) that can be seen as

[3] In the current implementation of Data Field Haskell this bound becomes wider than necessary. We expect to rectify this in later releases.

```
p_jacobi ad bd eps xd = iter (p_jacobi_iter ad bd) xd (p_conv eps)

p_jacobi_iter ad bd xd =
  forall p -> jacobi_iter (ad!p) (bd!p) (prUnion xd)

p_conv eps xd yd =
  maxnorm (forall p ->
    maxnorm ((forall i -> ((xd!p)!i - (yd!p)!i))
                          <\>(bounds (xd!p) `meet` bounds (yd!p))))
  < eps

prunion d1 d2 =
  forall x -> if (inBounds x (bounds d1)) then d1!x else d2!x

emptyfield = datafield (\x -> outofBounds) empty

prUnion = foldlDf prunion emptyfield
```

FIG. 4.3. *Data Field Haskell program for parallelised simulation.*

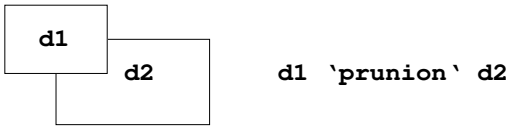

FIG. 4.4. `prunion`.

processor addresses. `p_jacobi` uses the same basic iteration pattern as the original specification, but now the entities are distributed: `ad` is a nested datafield of two-dimensional data fields, and `bd` and `xd` are nested one-dimensional data fields. Each iteration step is performed by `p_jacobi_iter`: it assembles a global array from the distributed array `xd`, which is then used by all processors for computing local updates of its segments of `xd` using its parts of `ad` and `bd`. `p_conv`, finally, computes the maximum norm of `xd` in parallel, by first computing local maximal differences for each segment of old and new `xd`, and then computing a global maximum of these.

The global array is assembled through `prUnion`, which "flattens" a distributed one-dimensional data field of one-dimensional data fields. It is a reduction over the binary operation `prunion` on data fields. This operation computes the "union" of two data fields giving priority to its first argument, see Fig. 4.4. If the data fields have disjoint bounds, then the operation acts somewhat like concatenation of lists. This similarity is not a mere coincidence, see Sect. 5.

The parallel algorithm above works on distributed data fields. The functions in Fig. 4.5 compute distributions. `divide a k lo hi` computes a data field of bounds that represents a `k`-partitioning of the interval $\{\texttt{lo}, \ldots, \texttt{hi}\}$. It partitions the rows of `a` as to make the numbers of elements of `a` in each partition as equal as possible, thus obtaining a static load balancing of the work. It is a divide-and-conquer algorithm, which itself is parallel. In each step, a point `midpoint` is selected between `lo` and `hi` such that the number of elements in rows $\texttt{lo} \ldots \texttt{midpoint}$ is as close to the number of elements in rows $\texttt{midpoint} + 1 \ldots \texttt{hi}$ as possible. The same procedure is then recursively applied until `k` intervals have been formed.

```
divide a 1 lo hi = datafield (\i -> (lo<:>hi)) (1<:>1)

divide a k lo hi =
  let mid = midsearch a lo hi
  in (divide a (k `div` 2) lo mid) `concat`
     (divide a (k - (k `div` 2)) (mid+1) hi)

midsearch a lo hi =
  midsearch_loc a lo hi lo hi (midpoint lo hi)
  where
    midsearch_loc a lo hi loloc hiloc mid
      | mid == hiloc || mid == loloc  = adjust a lo hi mid
      | slice_size a lo mid > slice_size a (mid+1) hi
        = midsearch_loc a lo hi loloc mid (midpoint loloc mid)
      | otherwise
        = midsearch_loc a lo hi mid hiloc (midpoint mid hiloc)

midpoint m n = m + ((n-m) `div` 2)

h_slice a b = a <\> (b<*>universe)

slice_size a lo hi = size (bounds (h_slice a (lo<:>hi)))

adjust a lo hi mid =
  if abs ((slice_size a lo (mid+1))-(slice_size a (mid+2) hi)) <
     abs ((slice_size a lo mid)-(slice_size a (mid+1) hi))
  then mid+1 else mid

concat d1 d2 = prunion d1 (forall x -> d2!(x-upperBound (bounds d1)))

run_p_jacobi a b eps x no_procs =
 let b_ac = bounds (matrix_curry a)
     dst  = divide a no_procs (lowerBound b_ac) (upperBound b_ac)
 in p_jacobi (distr_2 a dst) (distr_1 b dst) eps (distr_1 x dst)

matrix_curry a = forall i -> forall j -> a!(i,j)

distr_1 a b = forall p -> a <\> (b!p)

distr_2 a b = forall p -> h_slice a (b!p)
```

FIG. 4.5. *Code for computing a distribution of data fields, for distributing initial data, and for starting the parallel algorithm.*

`run_p_jacobi` uses the computed distribution to distribute initial data and start the parallel algorithm. It uses the help functions `matrix_curry`, which turns a two-dimensional data field into a data field of data fields, and `distr_1` and `distr_2`, which convert one- and two-dimensional data fields, respectively, into nested, partitioned data fields according to a partitioning specified by a data field of bounds.

5. A Formal Verification of the Correctness of the Distributed Jacobi Algorithm. We now consider how to verify formally that `p_jacobi` implements `jacobi` correctly. Following Theorem 2.1, we use the more abstract model of partial functions, with $\perp$ and $*$ identified, and rather than verifying directly that the implementation relation holds between the data fields we will define partial function versions *jacobi* and *p_jacobi* of them and prove that *p_jacobi* implements *jacobi*.

DEFINITION 5.1. *$f: A \to B$ is* implemented *by $g: C \to (D \to B)$ under (ϕ_0, ϕ_1) if $\forall x \in A. f(x) = g(\phi_0 x)(\phi_1 x)$.*

Definition 5.1 is tailored for implementations by curried functions, which model nested data fields. Now consider the case when f and g are recursively defined, say we have a simultaneous recursive definition $(f, g) = F(f, g)$ where F is a continuous function. The condition on f and g in Definition 5.1 is an *inclusive* (or *admissible*) predicate, which means we can use *Scott's fixed-point induction* to prove it [23].

PROPOSITION 5.2. *Define (f_*, g_*) as the least fixed-point of F, and let $(f_0, g_0) = (\perp, \perp)$, $(f_i, g_i) = F(f_{i-1}, g_{i-1})$ for $i > 0$. Then f_* is implemented by g_* under (ϕ_0, ϕ_1) if $\forall i > 0.(\forall x.(f_{i-1}(x) = g_{i-1}(\phi_0 x)(\phi_1 x)) \implies \forall x.(f_i(x) = g_i(\phi_0 x)(\phi_1 x)))$.*

Proof. Scott's fixed-point induction principle for inclusive predicates P states that for any nondecreasing chain $\{d_i\}_{i=0}^{\infty}$, $P(\bigsqcup_{i=0}^{\infty} d_i)$ holds if: $P(d_0)$ holds, and $\forall i > 0. P(d_{i-1}) \implies P(d_i)$. In our case, $\{(f_i, g_i)\}_{i=0}^{\infty}$ is a nondecreasing chain where $(f_0, g_0) = (\perp, \perp)$ and $\bigsqcup_{i=0}^{\infty}(f_i, g_i) = (f_*, g_*)$. We trivially have $\forall x. \perp(x) = \perp(\phi_0 x)(\phi_1 x)$ which means the base case holds. What remains to prove is the induction step, which is the stated condition in the proposition. □

We need some more results in order to prove that *p_jacobi* implements *jacobi*. `p_jacobi` uses data fields that are partitioned and distributed over processors. At certain points, they are reassembled from their parts using `prUnion`. The correctness of this reassembly depends on whether the distribution really is a true partitioning. We define partial function versions of `prunion` and `prUnion`:

$$f\ pr\ g = \lambda x. if\ (x \in dom(f), f(x), g(x))$$
$$Pr(f) = red(pr, f)$$

It is easy to see that pr is associative. Partitioning and distributing a data field corresponds to a particular kind of implementation:

DEFINITION 5.3. *$g: C \to (A \to B)$ is a* partitioning *of $f: A \to B$ under ϕ if f is implemented by g under (ϕ, id) and if $\phi(dom(f)) = dom(g)$.*

PROPOSITION 5.4. *If g is a partitioning of f under some ϕ, then $Pr(g) = f$.*

Proof. In general, it holds that $(Pr(g))(i) = g(p)(i)$ for some $p \in dom(g)$. Furthermore, $dom(Pr(g)) = \bigcup_{p \in dom(g)} dom(g(p))$. We have $g(\phi(i))(i) = f(i)$ for all $i \in dom(f)$, and $dom(g(\phi(i))) = \{ j \mid \phi(j) = \phi(i) \}$: thus, $\{ dom(g(\phi(i))) \mid i \in dom(f) \}$ is a (set) partitioning of $dom(Pr(g))$. Therefore, $(Pr(g))(i) = g(\phi(i))(i)$ for all $i \in dom(f)$, and thus $(Pr(g))(i) = f(i)$. We also have, for all $i \in dom(f)$, $dom(g(\phi(i))) = \{ j \mid j \in dom(f) \wedge \phi(j) = \phi(i) \}$. Since $\phi(dom(f)) = dom(g)$ every $p \in dom(g)$ is an image of some $i \in dom(f)$: thus, $dom(Pr(g)) = \bigcup_{i \in dom(f)} dom(g(\phi(i))) = dom(f)$. $Pr(g) = f$ follows. □

Second, we need to prove that under certain conditions the result of a nested reduction, over a distributed, partitioned data field, equals a straight reduction over the original data field. The latter result corresponds to the BMF "reduce promotion" law $\oplus/\oplus/l = \oplus/{+\!\!+}/l$ for nested lists l and associative operators $\oplus$ [1]. This law holds since concatenation of lists imposes a certain ordering on the elements of respective lists. $Pr(f)$ will behave as ${+\!\!+}/l$ if $dom(f(p)) \cap dom(f(q)) = \emptyset$ for $p \neq q$, and *if p*

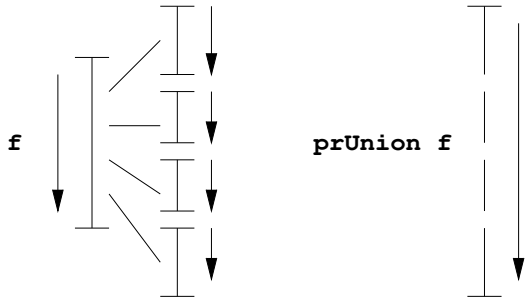

FIG. 5.1. *Compatible enumerations.*

being enumerated before q by $enum_f$ implies that each $x \in dom(f(p))$ is enumerated before each $y \in dom(f(q))$ by $enum_{Pr(f)}$.

DEFINITION 5.5. *$f: A \to (B \to C)$ has* compatible enumerations *if:*

- *$dom(f(x)) \cap dom(f(y)) = \emptyset$ when $x \neq y$,*
- *$enum_{Pr(f)}(n) = enum_{f(enum_f(m(n)))}(n - s(n))$, where $s(n)$, $m(n)$ are given by:*

$$
\begin{aligned}
s(n) &= \sum_{i<m(n)} size(f(enum_f(i))) \\
s(n) &\leq n \\
n &< s(n) + size(f(enum_f(m(n))))
\end{aligned}
$$

See Fig. 5.1. Here, the nth element of $Pr(f)$ belongs to the $m(n)$th element of f, and $s(n)$ is the sum of the sizes of the preceding elements of f. f having compatible enumerations basically means that the enumerations of each $f(i)$, $i \in dom(f)$, can be "concatenated", in the order given by the enumeration of f, to yield the enumeration of $Pr(f)$. This is similar to concatenating the elements in a list of lists. We can now state a law for curried partial functions, which corresponds to the BMF law for nested lists above:

THEOREM 5.6. *If f has compatible enumerations and $\oplus$ is associative, then $red(\oplus, \lambda x.red(\oplus, f(x))) = red(\oplus, Pr(f))$.*

Proof. (Sketch.) We first prove $red(\oplus, f \ pr \ g) = red(\oplus, f) \oplus red(\oplus, g)$ provided $dom(f) \cap dom(g) = \emptyset$, $enum_{f\,pr\,g}(n) = enum_f(n)$ when $0 \leq n < size(f)$, and $enum_{f\,pr\,g}(n) = enum_g(n - size(f))$ for $size(f) \leq n < size(f \ pr \ g)$. Then the result is proved by induction over the enumeration of f, using the definitions of Pr and red. □

Finally two simple results: the first was originally stated in [7], the second follows directly from the definition of *red*:

PROPOSITION 5.7. *If the n-ary function f is total and strict in all arguments, and if all g_i are total, $1 \leq i \leq n$, then $dom(\lambda x.f(g_1(x), \ldots, g_n(x))) = dom(g_1) \cap \cdots \cap dom(g_n)$.*

PROPOSITION 5.8. *If f and $\oplus$ are total, then $red(\oplus, f)$ yields a fully defined cpo element.*

Fig. 5.2 gives the definitions of *jacobi* and *p_jacobi*: cf. Figs. 4.1 and 4.3. We now prove that *p_jacobi* does provide a partitioning of *jacobi*, given that the inputs to the functions do.

$$f \setminus b = \lambda x.if(b(x), f(x), *) \quad \text{(explicit restriction)}$$

$$sum(f) = red(+, f)$$

$$iter(f, x, conv) = let\ \ x_{new} = f(x)\ \ in\ \ if(conv(x_{new}, x), x_{new}, iter(f, x_{new}, conv))$$

$$jacobi(a, b, \epsilon, x) = iter(\lambda x.jacobi\_iter(a, b, x), x, , conv_\epsilon)$$

$$jacobi\_iter(a, b, x) = \lambda i.(b(i) - sum(\lambda j.a(i, j) * x(j)) \setminus \lambda j.j \neq i))/a(i, i)$$

$$conv_\epsilon(x, y) = red(\max, \lambda i.\ |x(i) - y(i)|) < \epsilon$$

$$p\_jacobi(a', b'\epsilon, x') = iter(\lambda x'.p\_jacobi\_iter(a', b', x'), x', \epsilon, p\_conv_\epsilon)$$

$$p\_jacobi\_iter(a', b', x') = \lambda p.jacobi\_iter(a'(p), b'(p), Pr(x'))$$

$$p\_conv_\epsilon(x', y') = red(\max, \lambda p.red(\max, \lambda i.\ |x'(p)(i) - y'(p)(i)|)) < \epsilon$$

FIG. 5.2. *Definitions of* jacobi *and* p_jacobi.

THEOREM 5.9. *If: a, b, x are total; $dom(b) = dom(x) \subseteq \{1, \ldots, n\}$; $a(i, i) \neq 0, *$ for $1 \leq i \leq n$; a' is a partitioning of a under ϕ; b' and x' are partitionings of b, x, respectively, under ϕ', where $\forall j.\phi(i, j) = \phi'(i)$; and x' has compatible enumerations, then $p\_jacobi(a', b', \epsilon, x')$ is a partitioning of $jacobi(a, b, \epsilon, x)$ under ϕ'.*

Proof. Since *jacobi* and *p_jacobi* are nonrecursively defined in terms of *iter* we can perform the fixed-point induction over *iter*. We have

$$iter_n(f, x, conv) = let\ \ x_{new} = f(x)\ \ in\ \ if(conv(x_{new}, x), x_{new}, iter_{n-1}(f, x_{new}, conv)$$

Let $x_{new} = jacobi\_iter(a, b, x)$, $x'_{new} = p\_jacobi\_iter(a', b', x')$, and $conv = conv_\epsilon$. From the definition of $iter_n$ above and the definitions of *jacobi* and *p_jacobi*, we see that the result holds if: $conv_\epsilon(x_{new}, x) = p\_conv_\epsilon(x'_{new}, x')$, x'_{new} is a partitioning of x_{new} under ϕ', and $iter_{n-1}(\lambda x'.p\_jacobi\_iter(a', b', x'), x'_{new}, p\_conv_\epsilon)$ is a partitioning of $iter_{n-1}(\lambda x.jacobi\_iter(a, b, x), x_{new}, conv_\epsilon)$ under ϕ'. In the calls to $iter_{n-1}$, only x_{new} and x'_{new} have changed. Thus, by the fixed-point induction hypothesis, the partitioning relation between these calls holds if: x_{new} is total, x'_{new} is a partitioning of x_{new} under ϕ', and if x'_{new} has compatible enumerations. Note that this induction hypothesis is more complex than the one in Proposition 5.2: this is since we also need to prove the result about $conv_\epsilon$ and p_conv_ϵ, which are reductions. (However, it is easy to see that the base case still holds, so it suffices to prove the inductive step.) We now prove the required conditions in turn:

- x_{new} total: we have $x_{new} = jacobi\_iter(a, b, x) = \lambda i.(b(i) - sum(\lambda j.a(i, j) * x(j)) \setminus \lambda j.j \neq i))/a(i, i)$. Since all operations involved are total, and x, a, and b are total, this function is total.
- x'_{new} a partitioning of x_{new} under ϕ': we first prove that $dom(x_{new}) = dom(x)$ and $dom(x'_{new}) = dom(x')$, which then implies $\phi'(dom(x_{new})) = dom(x'_{new})$.
  - $dom(x_{new}) = dom(x)$: we have $dom(x) = dom(b)$. We now prove $dom(b) = dom(x_{new})$ which yields the result. Reconsider the definition

of x_{new} above. We have $dom(b) \subseteq \{1,\ldots,n\}$, $dom(\lambda i.sum(\cdots)) = U$ (the universal set), and $dom(\lambda i.a(i,i)) \supseteq \{1,\ldots,n\}$. Since the arithmetic operations are strict and total (also division when $a(i,i) \neq 0$), Proposition 5.7 yields $dom(x_{new}) = dom(b) \cap U \cap dom(\lambda i.a(i,i)) = dom(b)$.

- $dom(x'_{new}) = dom(x')$: since both b' and x' are partitionings of b and x under ϕ', we have $dom(b') = \phi'(dom(b)) = \phi'(dom(x)) = dom(x')$. Furthermore, $x'_{new} = \lambda p.jacobi\_iter(a'(p), b'(p), Pr(x'))$. Similar to above, we can check that $jacobi\_iter(a'(p), b'(p), Pr(x'))$ is defined exactly when $b'(p)$ is defined, which yields $dom(x'_{new}) = dom(b')$. Thus, $dom(x'_{new}) = dom(x')$.

It remains to show that $x_{new}(i) = x'_{new}(\phi'(i))(i)$ for all i. We have $x'_{new} = \lambda p.jacobi\_iter(a'(p), b'(p), Pr(x'))$. By Proposition 5.4 we have $Pr(x') = x$, thus $x'_{new} = \lambda p.jacobi\_iter(a'(p), b'(p), x)$. For any i we now have (since $\forall j.\phi(i,j) = \phi'(i)$):

$$\begin{aligned} x'_{new}(\phi'(i))(i) &= jacobi\_iter(a'(\phi(i,j)), b'(\phi'(i)), x)(i) \\ &= \cdots b'(\phi'(i))(i) \cdots a'(\phi(i,j))(i,j) \cdots x(j) \cdots a'(\phi(i,i))(i,i) \\ &= \cdots b(i) \cdots a(i,j) \cdots x(j) \cdots a(i,i) \\ &= jacobi\_iter(a,b,x)(i) \\ &= x_{new}(i) \end{aligned}$$

- x'_{new} has compatible enumerations: by the above, it holds that $dom(x'_{new}) = dom(x')$ and $\forall p.dom(x'_{new}(p)) = dom(x'(p))$ (the latter follows from the fact that x'_{new} partitions x_{new} and x' partitions x under the same ϕ', with $dom(x_{new}) = dom(x)$). Since x' has compatible enumerations, and compatible enumerations is a property only of the domains, the result follows.

It remains to prove $conv_\epsilon(x_{new}, x) = p\_conv_\epsilon(x'_{new}, x')$. Let $y = \lambda i.\,|x_{new}(i) - x(i)|$ and $y' = \lambda p.\lambda i.\,|x'_{new}(p)(i) - x'(p)(i)|$. If $red(\max, \lambda p.red(\max, y'(p))) = red(\max, y)$, then the equality holds. It is easy to see (through Proposition 5.7), that y', $y'(p)$, and y have the same domains as x', $x'(p)$ and x, respectively. Thus, y' has compatible enumerations, and by Theorem 5.6 $red(\max, \lambda p.red(\max, y'(p))) = red(\max, Pr(y'))$ since max is associative. It is also easy to see that y' is a partitioning of y under ϕ'. Proposition 5.4 then yields $Pr(y') = y$ which proves the equality. □

6. Related Work. An excellent survey of collection-oriented languages up to around 1990 is found in [20]. The computation of bounds for `forall`-abstraction yields the implicit intersection rule of FIDIL [19]. The arrays in FIDIL resemble data fields also in other respects, for instance they can have a wider variety of shapes than traditional array bounds.

Examples of functional data parallel and array languages are Connection Machine Lisp [22], Id [3], Sisal [4], NESL [2], Data Parallel Haskell [9], and pH [15]. These languages are intended for direct parallel implementation whereas Data Field Haskell targets the specification phase. Haskell itself [17] has also been suggested for data parallel programming [16]. FISh [11] is an imperative array language, which shares some features with Data Field Haskell such as advanced polymorphism. It is, however restricted to regular arrays and certain recursion patterns, which enables the generation of good code but makes it less suitable for specification of sparse or dynamic algorithms. A survey of the research in parallel functional programming is found in [8].

High-level specification and formal derivation of parallel programs has been considered in the BSP model [21] and in the Bird-Meertens Formalism [6]. Data Field Haskell could serve as a flexible format supporting these methods. Transformation of algorithm specifications in ML into programs for SIMD computers was considered in [5].

7. Conclusions and Further Research. We have presented the data field model and Data Field Haskell, a Haskell dialect that provides an instance of data fields. A possible application of Data Field Haskell is as a rapid prototyping tool for the early specification phase of parallel algorithms. We exemplified with the specification and initial development of a simple iterative linear equation solver, which could provide a starting point for the development of real production code. The initial specification was very close to the mathematical formulation, and the parallel version, once concieved, was straightforward to express with nested data fields. We made a formal verification that the parallel algorithm implements the specification through a partitioning of the data. A formal verification of this kind would probably be very hard to do for parallel programs written in current production languages.

The current implementation of Data Field Haskell could be improved in many ways, and it could certainly be given a parallel implementation. An elegant way to introduce explicit parallelism would be to use the type system of Haskell: "processor id" types could be introduced, with a 1-1-mapping to a given parallel architecture, and a parallelising compiler could then directly distribute the computation of data fields indexed with such types. An *efficient* parallel implementation would, however, require an efficient implementation of Haskell itself! Therefore, at least in a short perspective, we believe more in Data Field Haskell as a vehicle for specification of successively more refined parallel algorithms, which can serve as blueprints for actual parallel implementations.

We derived an interesting counterpart to a BMF law for reduction over curried partial functions. This line of investigation could certainly be pursued further, and also be extended to data fields. A difference is that lists always are ordered, while data fields only need to be ordered for certain operations. For instance, it is possible to formulate a version of Theorem 5.6 where the binary operation also is assumed commutative: then the requirement of compatible enumerations for the partial function can be dropped. There is clearly a rich variety of laws to explore for the subsequent use in program transformations and algorithm synthesis. The "added value" of such a theory over BMF would primarily be a greater ability to work with less ordered structures than lists, like indexed multidimensional structures. We believe this is beneficial for parallel algorithm design in application areas like scientific computing, where indexed structures prevail and the mapping phase to processors is important.

REFERENCES

[1] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, Berlin, 1987.

[2] G. E. Blelloch. Programming parallel algorithms. *Comm. ACM*, 39(3), Mar. 1996.

[3] K. Ekanadham. A perspective on Id. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. Addison-Wesley, 1991.

[4] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10:349–366, 1990.

[5] S. Fitzpatrick, T. J. Harmer, A. Stewart, M. Clint, and J. M. Boyle. The automated trans-

formation of abstract specifications of numerical algorithms into efficient array processor implementations. *Science of Computer Programming*, 28(1):1–41, 1997.

[6] S. Gorlatch. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming*, 33(1):1–27, 1999.

[7] 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.

[8] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[9] J. M. D. Hill. Data Parallel Haskell: Mixing old and new glue. Tech. Rep. 611, Queen Mary and Westfield College, Dec. 1992.

[10] J. Holmerin. Implementing data fields in Haskell. Technical Report TRITA-IT R 99:04, Dept. of Teleinformatics, KTH, Stockholm, Nov. 1999. `ftp://ftp.it.kth.se/Reports/paradis/DFH-report.ps.gz`.

[11] C. B. Jay and P. A. Steckler. The functional imperative: shape! In C. Hankin, editor, *Proc. 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Comput. Sci.*, pages 139–53, Lisbon, Portugal, Mar. 1998. Springer-Verlag.

[12] 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.

[13] B. Lisper. Data fields. In *Proc. Workshop on Generic Programming*, Marstrand, Sweden, June 1998. `http://wsinwp01.win.tue.nl:1234/WGPProceedings/`.

[14] B. Lisper and P. Hammarlund. The data field model. Preliminary version available as Tech. Rep. TRITA-IT R 99:02, Dept. of Teleinformatics, KTH, Stockholm, 1999. `ftp://ftp.it.kth.se/Reports/TELEINFORMATICS/TRITA-IT-9902.ps.gz`.

[15] R. S. Nikhil, Arvind, J. E. Hicks, S. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science, Jan. 1995.

[16] J. T. O'Donnell. Data parallelism. In Hammond and Michaelson [8], chapter 7, pages 191–206.

[17] 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/`.

[18] N. Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1995.

[19] L. Semenzato and P. Hilfinger. Arrays in FIDIL. In L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao, editors, *Arrays, Functional Languages, and Parallel Systems*, chapter 10, pages 155–169. Kluwer Academic Publishers, Boston, 1991.

[20] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, Apr. 1991.

[21] D. B. Skillicorn. Building BSP programs using the refinement calculus. External Technical Report TR96-400, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Oct. 1996.

[22] G. L. Steele and W. D. Hillis. Connection Machine LISP: Fine grained parallel symbolic programming. In *Proc. 1986 ACM Conference on LISP and Functional Programming*, pages 279–297, Cambridge, MA, 1986. ACM.

[23] G. Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1993.