Data Field Haskell

Jonas Holmerin[1] and Björn Lisper[2]

[1] Department of Numerical Analysis and Computing Science, Royal Institute of Technology, SE-100 44 Stockholm, SWEDEN
joho@nada.kth.se
[2] Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, SE-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 Haskell dialect that provides an instance of data fields. It can be used for very generic collection-oriented programming, with a special emphasis on multidimensional structures. We give a brief description of the data field model and its underlying theory. We then describe Data Field Haskell, and an implementation.

1 Introduction

Indexed data structures are important in many computing applications. The canonical indexed data structure is the array, but other indexed structures like hash tables and explicitly parallel entities are also common. In many applications the indexing capability provides an important part of the model: when solving partial differential equations, for instance, the index is often closely related to a physical coordinate, and explicitly parallel algorithms often use processor ID's as indices.

Since the time of APL [5] it has been recognised that a programming model that provides operations directly on data structures can be very convenient. This style of programming is often called *collection-oriented programming* [27]. Modern array and data parallel languages like Fortran 90 [3] provide support for this programming style, as do higher-order functional languages, which usually offer collection-oriented list operations.

However, these languages typically restrict the scope of collection-oriented operations to a single kind of data type, and the semantics of these operations can be somewhat ad-hoc. Data fields[1] model indexed data structures as partial functions supplied with explicit information about their domains. This leads to a programming model that is highly uniform over different indexed data structures, and where the operations are designed according to common semantical principles.

Data Field Haskell is a Haskell dialect where the arrays have been replaced by an instance of data fields. This particular instance consists of multidimensional array-like data fields, which can be sparse, dense, or sparse in some dimensions and dense in others. Thus, this dialect is targeted towards rapid prototyping of parallel algorithms, which may involve sparse structures, but we believe it is useful for a wide range of applications amenable to collection-oriented programming.

There is reason to believe that, in a few years' time, the advances in semiconductor technology will force the replacement of current processor architectures with multiprocessors on a chip [7]. When this happens, parallelism will become central also in mainstream computing. The collection-oriented paradigm provides an attractive parallel programming model due to its conceptual simplicity. This motivates a

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

continued investigation in general collection-oriented programming models, and is one reason for the development of Data Field Haskell.

The rest of this paper is organised as follows. Section 2 gives a brief description of the underlying data field model. Section 3 describes Data Field Haskell, and Section 4 gives a simple example of its use. Section 5 contains a description of the current implementation. Section 6 provides an account for related work. In Section 7, finally, the story is wrapped up. The limited space does not allow a complete description of Data Field Haskell here – see [1, 11] for the details.

Various versions of the data field model have been described elsewere [8, 16–18]. The use of Data Field Haskell for rapid prototyping of parallel algorithms has been reported in [12, 19]. The contribution of this paper is a more thorough description of the language and of an implementation.

2 The Data Field Model

The concept of data fields is based on the more abstract model of indexed data structures as functions with finite domain [8, 16]. An array with range `[1..n]`, for instance, can be seen as a function from $\{1, \dots, n\}$, but we could also model "irregular" indexed structures as functions with non-contiguous, possibly non-numerical domains. In order to give partial functions conventional function types they are seen as functions that return a distinguished *error value* $*$, with algebraic properties similar to $\perp$, when called with an argument outside their domains.

The partial function model is simple and powerful, and most types of collection-oriented operations [27] can be defined as higher order functions operating on partial functions [8, 18]. However, certain operations require explicit information about the function domains. Thus, we consider entities (f, b) – the data fields – where f is a function and 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 an interpretation as a predicate (or set).
- 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").
- The bounds *all* and *nothing* 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, without revealing the inner structure of the bounds. They must have certain properties, see [18].

The theory of data fields also defines *φ-abstraction*, a syntax for convenient definition of data fields that parallels λ-abstraction for functions. The meaning of $\varphi x.t$ is a data field $(\lambda x.t, b)$ where b provides an upper approximation to the domain of $\lambda x.t$. The purpose of φ-abstraction is to provide a formal semantics for collection-oriented operations where the bound of the result is implicitly given by the bounds of the operands. Such operations are convenient to use and common in array languages, and the data field model extends them beyond arrays.

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 [17, 18]. The new data types are `Datafield a b` for sparse/dense array datafields and `Bounds a` for the corresponding bounds. `a` must belong to the classes `Ix` (array index types) and `Pord` (types with partial ordering, which is convenient when defining certain operations on bounds: see [11]). `Pord` has the same instances as `Ix`: thus, possible index types for data fields are the same as for Haskell arrays (integers, characters, enumerations, and single-constructor data types whose components are index types). We will omit qualifications "`(Pord a,Ix a) =>`" when they are evident.

3.1 Basic Operations on Data Fields

`datafield` builds data fields from functions and bounds, and `bounds` provides the bounds of a data field:

```
datafield :: (a -> b) -> Bounds a -> Datafield a b
bounds :: Datafield a b -> Bounds a
```

As for Haskell arrays, the infix operation `!` is used for indexing. The constant `outofBounds` represents $*$, and the predicate `isoutofBounds` tests for this value.

3.2 Bounds

Data Field Haskell has a rich variety of bounds. They are classified as either *finite* or *infinite*. There are a number of operations to construct them:

`(<:>) :: a -> a -> Bounds a` yields *dense bounds*, i.e., usual array bounds. For instance, `(1,1)<:>(10,20)` returns a bound representing the rectangle with lower left corner $(1,1)$ and upper right corner $(10,20)$. Dense bounds are finite.

`sparse :: [a] -> Bounds a` creates *sparse bounds* that represent general finite sets. `sparse [(1,2),(17,9),(1,2),(42,44)]`, for instance, returns a sparse bound representing $\{(1,2),(17,9),(42,44)\}$. Sparse bounds are also finite.

`predicate :: (a -> Bool) -> Bounds a` forms *predicate bounds*. For instance, `predicate (\x -> f x /= 0)` represents the set where the function `f` is nonzero. Predicate bounds are classified as infinite.

`universe` (infinite) represents the universal set (the bound *all*) and `empty` (finite) the empty set (the bound *nothing*).

`(<*>) :: Bounds a -> Bounds b -> Bounds (a,b)` defines *product bounds* representing Cartesian products. It can be used to create conventional multidimensional array bounds, e.g.,`(1<:>10)<*>(1<:>20)` (which equals `(1,1)<:>(10,20)`), but also other bounds like `(sparse [5,7,13])<*>(1<:>10)` and `(1<:>10)<*>universe`. `b1<*>b2` is finite precisely when both `b1` and `b2` are. In general, `prod_`n forms n-tuples of bounds[2]. Some two-dimensional bounds are illustrated in Fig. 1.

3.3 Operations on Bounds

The most basic operations on bounds are `join` and `meet` ($\sqcup$ and $\sqcap$). They essentially compute the union and intersection, respectively, of their arguments seen as sets (if the arguments are both dense, then `join` may compute an overapproximation of the union, see Fig. 2). The kind of bound computed depends on the arguments as shown in Table 1. `join` and `meet` for product bounds are defined elementwise, i.e., the equation

```
(bx1<*>by1) `meet` (bx2<*>by2) = (bx1 `meet` bx2)<*>(by1 `meet` by2)
```

holds for `meet` and similarly for `join`.

[2] `<*>` is syntactic sugar for `` `prod_2` ``.

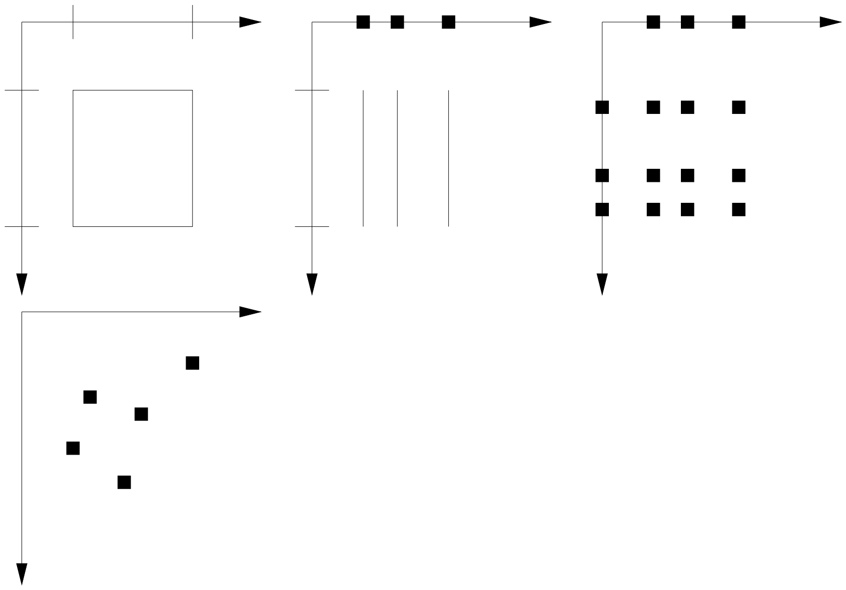

Fig. 1. Some two-dimensional bounds: three product bounds, and a sparse two-dimensional bound.

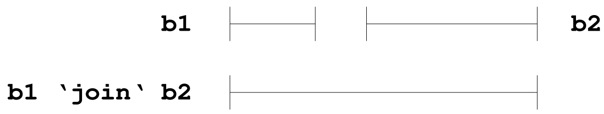

Fig. 2. Join of two one-dimensional dense bounds.

`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 1. Result "types" of `join` and `meet` as a function of the argument "types". 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.

`join` and `meet` are used primarily to define higher level data field constructs. An example is the explicit restriction operator on bounds, `<\>`. It satisfies the following equation:

```
(datafield f b1) <\> b2 = datafield f (b2 `meet` b1)
```

We now see why infinite bounds can make sense: for instance,

```
(datafield f (1<:>n)) <\> predicate p
```

will yield a sparse datafield, defined for the points in the range `1..n` where `p` is true. This provides a data field counterpart to array operations that are performed for the indices where a "mask" is true [15]. Some more operations on bounds are:

- `finite :: Bounds a -> Bool`, which tests bounds for finiteness,
- `enumerate :: Bounds a -> [a]`, which returns the list of elements of the set defined by a finite bound in the following order: for finite non-product bounds in the order given by the "<" operation in the `Ord` class, and for product bounds in the lexicographic order defined by the orders of the components.
- `size :: Bounds a -> Int`, which gives the number of elements in a finite bound, and
- `inBounds :: a -> Bounds a -> Bool`, which checks for membership in the set defined by a bound.

3.4 Operations on Finite Data Fields

Sometimes it is desirable to force the evaluation of all elements in a data field. There are, for instance, parallel algorithms whose efficiency depends on the compile-time knowledge of which computations to perform. This is similar to strictness declarations for functions, which sometimes are necessary to ensure efficient execution. To this end, we have defined three *data field evaluators*, all of type

```
(Pord a, Ix a, Eval a) => Datafield a b -> Datafield a b
```

that evaluate their respective arguments to different degrees. `hstrictTab`, for instance, evaluates all elements in a hyperstrict fashion (i.e., to the innermost constructor).

`foldlDf`, of type

```
(Pord a, Ix a, Eval a) => (b -> c -> b) -> b -> (Datafield a c) -> b
```

is the data field equivalent to `foldl` for lists. It reduces its data field argument in the order given by the enumeration of its bound. The reduction only includes the values indexed by elements in the domain of the corresponding partial function (note that the bound may overapproximate this domain: see [11] for details). As for lists, there are various versions of data field folds [11].

The operations in this section are only meaningful for finite data fields and will yield a runtime error if applied to an infinite data field.

3.5 Forall-abstraction

Data Field Haskell provides a form of φ-abstraction, with the following syntax (described in the metasyntax of the Haskell report [23]):

$$\texttt{forall}\ apat_1\ \ldots\ apat_n\ \texttt{->}\ exp$$

Thus, the syntax is analogous to λ-abstraction in Haskell and includes such features as pattern-matching (which is convenient when defining multidimensional data fields). Type inference works in the same way as for λ-abstraction, although the identifiers being abstracted over must be instances of the `Pord` and `Ix` classes. The semantics of `forall`-abstraction is

```
forall x -> t = datafield (\x -> t) b
```

where the bound `b` is a function of the form of `t`.

The limited space prohibits a detailed account for how `b` is computed: the exact rules are found in [1, 11]. Here, we give an informal description supported by representative examples. First, if `a!x` occurs in a strict position in the body of `forall x -> ...` then `bounds a` should constrain the bounds of `forall x -> ...`. Thus,

```
bounds (forall x -> a!x + b!x + 17) = (bounds a) `meet` (bounds b)
```

The principle generalises to `forall`-abstraction over tuples, which should have product bounds where each component constrains the respective variable in the tuple. Thus,

```
bounds (forall (x,y) -> a!x * b!y) = (bounds a) <*> (bounds b)
```

so this expression yields the outer product of `a` and `b` with the expected bounds.

For conditionals, any of the branches could be taken for any value of `x`. Thus, the bounds from the branches should be joined. Moreover, the conditional is strict in the condition, thus,

```
bounds (forall x -> if a!x then b!x else c!x) =
  (bounds a) `meet` ((bounds b) `join` (bounds c))
```

Multidimensional arrays are important in array languages, and they often provide convenient syntax to select subarrays from matrices. In order to generalise this feature to data fields, components of product bounds of multidimensional data fields occurring in `forall`-abstraction can constrain the bound of the abstraction. Thus, if `bounds a = b1<*>b2`, we have[3]

```
bounds (forall x -> a!(1,x)) = b2
```

(selection of row one), and

```
bounds (forall x -> a!(x,x)) = b1 `meet` b2
```

(main diagonal). This feature can be combined with `forall`-abstraction over tuples, like

```
bounds (forall (x,y) -> a!(y,x)) = b2 <*> b1
```

("data field transpose"). If the bound of `a` is a sparse multidimensional bound, then the smallest enclosing product bound is first computed and the above then applies.

Finally, we allow translations of bounds w.r.t. linear offsets, e.g., if `bounds a = 1<:>5` then

```
bounds (forall x -> a!(x+1)) = 0<:>4
```

Sparse bounds are translated similarly, and this feature combines with the others. If none of the previous cases apply (e.g., `forall x -> a!(f x)`), then the bound `universe` will result.

The "compute bounds first" evaluation order of `forall`-abstraction gives data fields a lazy flavour. For instance, one may define a two-dimensional data field with finitely many infinitely long columns; rows are then still finite data fields.

[3] a more exact bound would be `if (inBounds 1 b1) then b2 else empty`, but the current version of Data Field Haskell does not compute this.

3.6 For-abstraction

`for`-abstraction provides a convenient syntax to define data fields by cases. It essentially defines a data field from a list of pairs of bounds and expressions and can be thought of as a "parallel case" where the different bounds provide the cases. The syntax is

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

with semantics

(forall *pat* -> if inBounds *pat* (e_1) then e'_1 else if ...
else if inBounds *pat* (e_n) then e'_n else outofBounds) <\>
(e_1) ‘join‘ (e_2) ‘join‘ ... ‘join‘ (e_n)

4 A Simple Example

The limited space only allows a short example, see [1, 12, 19] for more examples. Consider the linear equation system $Ax = b$, where A is an $n \times n$ lower-triangular matrix. (1) gives the classical forward-solving algorithm for computing x:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}, \quad i = 1, \ldots, n \tag{1}$$

This algorithm can be more or less directly expressed in Data Field Haskell:

```
dfSum = foldlDf (+) 0

fsolv a b = forall i ->
  (b!i - dfSum (for j in 1<:>(i-1) -> a!(i,j) * (fsolv a b)!j))
  /a!(i,i)
```

Note how "`dfSum (for j in 1<:>(i-1) -> ...)`" corresponds to "$\sum_{j=1}^{i-1} \ldots$".

What is the bound of `fsolv a b`? It will be constrained by the bound of `b`, and the bounds with respect to `i` derived from `dfSum (...)` and `a!(i,i)`. If `bounds a = b1<*>b2`, then the latter bounds are `b1` and `b1 ‘meet‘ b2`, respectively, and we obtain

```
bounds (fsolv a b) = (bounds b) ‘meet‘ b1 ‘meet‘ b1 ‘meet‘ b2
```

If `(bounds b) = b1 = b2 = 1<:>n`, then `bounds (fsolv a b) = 1<:>n` as expected. The bound of the data field being summed over, finally, is given by the constraints on `k`: thus, it equals `1<:>(i-1) ‘meet‘ b2 ‘meet‘ bounds (fsolv a b)`. With `bounds b`, `b1`, and `b2` as above this equals `1<:>(i-1)`.

Interestingly, the code above works also for sparse `a`: a sparse version of a dense matrix can be created with the very generic function `sparsify` defined below:

```
sparsify x = x <\> predicate (\i -> x!i /= 0)
```

If `x` has a finite bound, then `sparsify x` will have a finite sparse bound. If `fsolv` is given a sparse `a`, the current version of Data Field Haskell will first create the bounds `b1` and `b2` by *projecting* `bounds a` as indicated in Fig. 3, and then the above works as before. Note that this leads to loose approximations: in particular for each `i`, the bounds for the summed data field really only needs to contain the `k` in `1<:>(i-1)` where `a(i,k)` is defined. It is possible to define a more complex scheme for deriving constraints of bounds arising from the use of sparse multidimensional data fields, which yields exactly this: the details can be found in [18]. However, Data Field Haskell does not yet use this scheme.

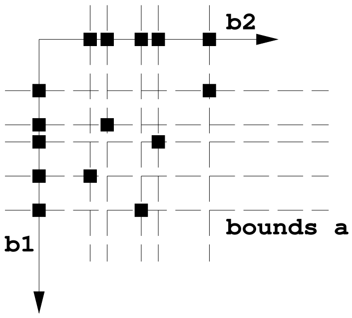

Fig. 3. The two one-dimensional projections of a sparse, two-dimensional bound.

5 Implementation

Our implementation of Data Field Haskell is based on the NHC compiler [25], which implements Haskell v. 1.3. The execution mechanism is graph reduction, which is performed by a variant of the G-machine. Our implementation consists of:

- Modifications to the front-end in order to parse and type-check `forall` and `for`-abstractions,
- automatic derivation of instances for the new type class `Pord`, and for the `Eval` class which has been slightly modified [11],
- a program transformation of intermediate code with `forall`- and `for`-abstractions into intermediate code without `forall` and `for`-abstractions,
- the abstract data types for `Datafield` and `Bounds` implemented in Haskell, and
- simple exception handling (used to implement `outofBounds`), implemented mostly by modifications to the back-end.

Portability and development time was deemed more important than execution speed, thus we have strived to make most of the implementation in Haskell itself. We have not implemented any advanced optimizations.

The front-end modifications are quite straightforward, as the automatic derivation of instances for the `Pord` and `Eval` classes. `for`- and `forall`-abstractions are translated into intermediate code that uses the `datafield` function to build data fields. In this process, calls to `join` and `meet` are also introduced. These operations obey the following equations, and we perform the corresponding simplification of expressions for bounds in the translation:

$$\texttt{universe}\ \texttt{`meet`}\ x = x \qquad \texttt{empty}\ \texttt{`join`}\ x = x$$
$$x\ \texttt{`meet`}\ \texttt{universe} = x \qquad x\ \texttt{`join`}\ \texttt{empty} = x$$

The implementation of the abstract data types for data fields and bounds was not entirely straightforward to do in Haskell. The problem is `Bounds a`. Ideally, one would define this as an algebraic data type with constructors for the different kinds of bounds. However, product bounds do not fit into this scheme since they require that `a` is a tuple type. It would indeed be possible to define a type `PBounds_n a1 ... an = ...` that includes product bounds, but this type could then not be used for bounds over non-tuple-types and one would have to use different types for bounds and data fields over tuple types and non-tuple types. Overloading the operations on data fields and bounds through the class system does not work, since the type constructors `Bounds` and `PBounds_n` have different arities. Pattern-matching in type declarations, like

```
data PBounds_n (a1,...,an) = ...
```

would make it possible to define a constructor class for bounds, but this is not allowed in Haskell.

Thus, we have reverted to a low-level implementation of data fields and bounds, done in Haskell but with incorrect types. The implementation has some similarities with how dictionaries are used to implement overloading in Haskell. Coercion functions, which are manually given (incorrect) function types, are used as interfaces between the `Datafield` and `Bounds` types and their implementations.

Sparse bounds and tabulated data fields are represented by an abstract data type for sets, which is based on balanced binary trees. If n is the number of elements stored in the tree, then membership tests (and lookups) are done in time $O(\log n)$, unions, intersections, enumerations, and folds in time $O(n)$, and the size is calculated in time $O(\log^2 n)$.

The production of ordinary error values in Haskell results in immediate termination. `outofBounds` must be handled in a less strict fashion, since data fields represent partial functions where the bounds may overapproximate the partial function domain, and certain operations should only be performed over the elements in this domain. Thus, it must be possible to just skip occurrences of `outofBounds` rather than terminating directly when it appears.

We wanted the implementation of this to be reasonably efficient. Therefore we have introduced a simple exception handling mechanism. On the Haskell kernel level a function `handle` is introduced that adheres to the following:

```
handle x y = y  -- if x evaluates to outofBounds
handle x y = x  -- otherwise
```

`isoutofBounds` can now be defined as:

```
isoutofBounds = handle (seq x False) True
```

`handle` is implemented by catching exceptions, and `outofBounds` is implemented by throwing them.

$< n_1 : n_2 : S, G, \texttt{HANDLE} : C, D, E >$
$\Rightarrow < n_1 : S, G, \texttt{EVAL} : \texttt{REMOVEHANDLER} : C, D, (n_2, S, C, D) : E >$

$< S, G, \texttt{REMOVEHANDLER} : C, D, t : E > \Rightarrow < S, G, C, D, E >$

$< S, G, \texttt{FAIL} : C, D, (n, S', C', D') : E > \Rightarrow < n : S', \texttt{EVAL} : C', D', E >$

Fig. 4. State transitions for `HANDLE` , `REMOVEHANDLER` and `FAIL`.

The exception handling was implemented by modifying the G-machine of NHC. The basic G-machine, as described in [24], has four-tuples $< S, G, C, D >$ as states. Here, S is a stack of *node names*, G is the graph, C is the sequence of G-code being executed, and D is the *dump*, a stack of pairs of code sequences and stacks. The G-machine of NHC adheres to this scheme, although its instruction set and low-level representations are somewhat different. Our modified G-machine has five-tuples $< S, G, C, D, E >$ as states. The new component E, the *exception stack*, consists of quadruples (n, S, C, D) of a node name, a stack, a code sequence and a dump. (S, C, D) saves the current state when the handling of an exception is set up, and n points to the node to be evaluated on failure. We also need three new instructions: `HANDLE`, `REMOVEHANDLER`, and `FAIL`. The code generated for `outofBounds` is simply

```
FAIL
```

and the code for `handle x y` is

```
<code that puts x on the stack>
<code that puts y on the stack>
HANDLE
```

The idea is to abort the evaluation of `x` if `FAIL` is executed, restore the machine state to what is was before the evaluation of `x` began, and evaluate `y`. The semantics of the instructions as transitions of the modified G-machine is shown in Figure 4.

The description above is for exception handling in the basic G-machine. Our actual solution for the G-machine of NHC is slightly different, due to the internal details of this G-machine, but the basic idea is the same. See [11].

6 Related Work

There is a wealth of collection-oriented languages and it is impossible to give a full account here. An excellent survey of collection-oriented languages up to around 1990 is found in [27]. Array and data parallel languages like Fortran 90, HPF [15], and *lisp [29] have been important sources of inspiration for Data Field Haskell. The language closest to Data Field Haskell is probably FIDIL [26], whose *implicit intersection rule* corresponds to the propagation of bounds from strict positions below a `forall`-abstraction. 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 [28], Id [4], Sisal [6], NESL [2], Data Parallel Haskell [10], and pH [21]. These languages are intended for direct parallel implementation whereas Data Field Haskell targets collection-oriented programming in general, with more emphasis on expressiveness than efficiency. Haskell itself [23] is to some extent collection-oriented through its set of collective list operations, and it has been suggested for data parallel programming [22]. FISh [13] 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 [9].

"Bulk types", like the ones provided by the STL C++ library [20], provide generic collection-orientation and are similar in this respect to data fields. Peyton Jones [14] has used the class system of Haskell to define bulk types. Bulk types do not provide any particular support for multidimensional structures, and there is no counterpart to `forall`-abstraction and implicit derivation of bounds for expressions.

7 Conclusions and Further Research

We have defined and implemented Data Field Haskell, a Haskell dialect where data fields replace arrays. Data fields are designed with the abstract view of indexed structures as partial functions in mind. This leads to the view of bounds as set representations, and to the design of `forall`-abstraction, which is inspired by λ-abstraction. The intention has been to create a language that supports collection-oriented programming at a very high level. Although our initial inspiration comes from array and data parallel programming, we believe that the data field concept is general enough to support collection-oriented programming in a variety of applications.

Data Field Haskell is designed for expressiveness rather than speed. We believe this is the right place to start, and then investigate how restricted sublanguages can be given an efficient implementation and how performance-enhancing features like mutable data fields could be introduced. Parallel implementations are also certainly possible.

The efficiency of our current implementation can also be greatly improved. We have furthermore found some cases of `forall`-abstraction where it would be natural to have a tighter bound. We plan to upgrade our implementation to Haskell 98: in this process, we may fix some of the current deficiencies.

Another desirable feature is *elemental intrinsics overloading*, which refers to the ability in some array languages to apply certain "scalar" operators to arrays with the meaning that it is applied to each element. For data fields, it would be natural to resolve this overloading into `forall`-expressions, e.g., `a+b` $\rightarrow$ `forall x-> a!x + b!x` provided that `a` and `b` have the proper data field type. To some extent this is possible to do within the class system of Haskell, but the resulting overloading has certain restrictions and is also likely to lead to inefficiencies. We are investigating another scheme for elemental intrinsics overloading that is less restricted, but it is still only defined for explicitly typed languages [30]. An obvious goal is to extend this scheme to implicitly typed languages.

The low-level representation of data fields and bounds is somewhat unsatisfactory, since it hurts the portability of the implementation. If Haskell's algebraic type declarations allowed pattern matching on type parameters then it would be possible to define classes for bounds and data fields. We could then do away with the low level representations. This would also make it possible for users to define their own types of bounds. The formal data field model [18] was specifically designed to support the development of abstract data types for bounds and data fields, and the ability to define new types of bounds would be an important enhancement of the language.

References

1. Data Field Haskell homepage. `http://www.it.kth.se/labs/paradis/dfh/`.
2. Guy E. Blelloch. Programming parallel algorithms. *Comm. ACM*, 39(3), March 1996.
3. Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to FORTRAN 90*. Programming Languages. McGraw-Hill, 1990.
4. Kattamuri Ekanadham. A perspective on Id. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. Addison-Wesley, 1991.
5. A.D. Falkoff and K.E. Iverson. The Design of APL. *IBM Journal of Research and Development*, pages 324–333, July 1973.
6. John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10:349–366, 1990.
7. Tom R. Halfhill. Sun reveals secrets of "magic". *Microprocessor Report*, pages 13–17, August 1999.
8. Per Hammarlund and Björn 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.
9. Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
10. Jonathan M. D. Hill. Data Parallel Haskell: Mixing old and new glue. Tech. Rep. 611, Queen Mary and Westfield College, December 1992.
11. Jonas Holmerin. Implementing data fields in Haskell. Technical Report TRITA-IT R 99:04, Dept. of Teleinformatics, KTH, Stockholm, November 1999. `ftp://ftp.it.kth.se/Reports/paradis/DFH-report.ps.gz`.

12. Jonas Holmerin and Björn Lisper. Development of parallel algorithms in Data Field Haskell. Accepted to Euro-Par 2000, 2000.
13. C. Barry Jay and P. A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Proc. 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Comput. Sci.*, pages 139–53, Lisbon, Portugal, March 1998. Springer-Verlag.
14. Simon Peyton Jones. Bulk types with class. In *Electronic Proceedings of the 1996 Glasgow Functional Programming Workshop*, Ullapool, July 1996.
15. Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1994.
16. Björn Lisper. Data parallelism and functional programming. In Guy-Reneé Perrin and Alain 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, March 1996. Springer-Verlag.
17. Björn Lisper. Data fields. In *Proc. Workshop on Generic Programming*, Marstrand, Sweden, June 1998. `http://wsinwp01.win.tue.nl:1234/WGPProceedings/`.
18. Björn Lisper and Per Hammarlund. The data field model. Submitted. Preliminary version avaliable as Tech. Rep. TRITA-IT R 99:02, Dept. of Teleinformatics, KTH, Stockholm, 2000.
19. Björn Lisper and Jonas Holmerin. Development and verification of parallel algorithms in the data field model. In Sergei Gorlatch and Christian Lengauer, editors, *Proc. 2nd Int. Workshop on Constructive Methods for Parallel Programming*, pages 115–130, Ponte de Lima, Portugal, July 2000.
20. David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, 1996.
21. Rishiyur S. Nikhil, Arvind, James E. Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science, January 1995.
22. John T. O'Donnell. Data parallelism. In Hammond and Michaelson [9], chapter 7, pages 191–206.
23. John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell: A non-strict purely functional language, version 1.4, April 1997. `http://www.haskell.org/definition/`.
24. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1987.
25. Niklas Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1995.
26. Luigi Semenzato and Paul Hilfinger. Arrays in FIDIL. In Lenore M. R Mullin, Michael Jenkins, Gaétan Hains, Robert Bernecky, and Guang Gao, editors, *Arrays, Functional Languages, and Parallel Systems*, chapter 10, pages 155–169. Kluwer Academic Publishers, Boston, 1991.
27. Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, April 1991.
28. Guy 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.
29. Thinking Machines Corporation, Cambridge, MA. *Getting Started in *Lisp*, June 1991.
30. Claes Thornberg. *Towards Polymorphic Type Inference with Elemental Function Overloading*. Licentiate thesis, Dept. of Teleinformatics, KTH, Stockholm, May 1999. Research Report TRITA-IT R 99:03.