Data Fields

Björn Lisper
Dept. of Teleinformatics, KTH
Electrum 204
S-164 40 Kista, Sweden
lisper@it.kth.se

May 15, 1998

Abstract

This position paper describes the data field model, a general model for indexed data structures. The aim of this model is to capture the essence of the style of programming where computing on data structures is expressed by operations directly on the structures rather than operations on the individual elements. Array and and data parallel languages support this programming style, and functional languages often provide second order operations on lists and other data structures for the same purpose. The data field model is designed to be abstract enough to encompass a wide range of explicitly or implicitly indexed structures. Thus, algorithms which are expressed in terms of data fields and general operations on them will be independent of the choice of structure from this range – i.e., generic w.r.t. this choice. This means that the data field approach has some in common with polytypic programming and the theory of shapes.

1 Introduction

Many computing applications require indexed data structures, i.e., a collection of uniformly typed data which can be (directly or indirectly) indexed in order to retrieve a certain value. Examples are homogenous lists, arrays, data parallel entities, trees, and nested sequences to mention a few. The indexing capability need not be explicit (like, for instance, when representing a set by a list), but in many applications it provides an important part of the model. Examples of the latter is when solving partial differential equations (PDE's), where the index is closely related to a physical coordinate, in image and signal processing, and in linear algebra.

Since the time of APL [5] it has been recognized that a programming model which provides operations directly on data structures can be very convenient. The data parallel programming model [10] is an exponent of this style (albeit often with a close connection to some underlying parallel architecture providing support for these operations). Modern array- and data parallel languages like Fortran 90 [3], High Performance Fortran (HPF) [7], NESL [2], and Sisal [6, 17] provide support, as well as many functional languages which have a rich set of list operations corresponding to the array- and data parallel operations.

Most of these languages have in common, though, that they restrict the scope of direct operations on data structures to a single data type or class of data types, such as arrays or lists. But clearly there are some underlying patterns which recur throughout. The data field model is an attempt to capture the essence of these by providing an general programming model for indexed structures, whose design is based on the more abstract view of indexed structures as partial functions. The goal is to provide a model with a uniform semantics over a wide range of structures, which is flexible, expressive and not boxed in by the idiosyncrasies of a certain kind of structure.

Some papers on the data field model have been published. The early [9] gives an account of the data-structure-as-partial-function view and uses an informal metalanguage for partial functions to define common operations on data structures as operations on partial functions. In [14] an abstract interpretation called *extent analysis* is developed, which can find the domain of definition for certain recursively defined partial functions. A tutorial on the early data field model of indexed data structures as partial functions is found in [13]. This position paper gives a short account for the present data field model which is presented in full detail in [15].

1.1 Operations on indexed data structures

Which operations on indexed data structures are there, then? One can distinguish five major groups of operations which occur throughout.

Elementwise applied operations apply a "scalar" operation f to every element in a data structure A. The canonical example is the `map` operation on lists. Languages with explicitly indexed structures typically provide elementwise applied n-ary operations: for instance, Fortran 90 allows constructs like `X+Y` which adds the elements of the arrays `X` and `Y` elementwise. Here, the indexing is used to find out which elements from `X` and `Y` to add.

A second group of operations reorder indexed data structures. This can be seen as *communication* where elements of data structures are transferred between index locations. *Parallel read* from a data structure A yields a new data structure whose value at index i is $A(\mathit{source}(i))$. Here, *source* is some function from indices to indices. *Parallel write* concurrently sends, for all indices i, the element with index i to $\mathit{dest}(i)$. This is a potentially nondeterministic operation, but it can also be given a deterministic semantics in various ways.

The third group of operations perform some kind of *replication*. For instance, The Sisal operation `array_fill` creates an array of copies of a given value. Array languages often support the replication of arrays into arrays of higher dimensions, e.g., replicating a vector into a matrix with copies of the vector as columns.

Restriction operations apply a boolean condition elementwise, as a "mask", in order to select a part of a data structure. For instance, in Fortran 90,

```
WHERE (A < 0.0) A = -A
```

effectively sets every element of the array `A` to its absolute value. For lists, restriction is provided by `filter` operations.

Reduction operations, finally, compute some single value as a function of the elements of a data structure. Usually, the function is composed of some repeatedly applied binary operation. Examples are the `foldl` and `foldr` operations on lists.

Finally, a note on syntax for these operations. The most direct approach is of course to provide explicit operations, like `map` for elementwise applied operations. But other syntaxes exists and have proven to be useful. Array languages like Fortran 90 often provide *elemental intrinsics*, a number of scalar operations which also are overloaded as elementwise applied operations on arrays. The elementwise addition of `X` and `Y` above is an example. A third form of notation "quantifies" over a given range of indices, e.g., in Sisal,

```
for i in 1,n returns array of X[i]+Y[i]
```

adds `X` and `Y` elementwise for the indices `1` to `n`. The imperative `FORALL` statement [1] in HPF is very related.

1.2 The Data Field View of Indexed Data Structures

Data fields are pairs (f, b) where f is a function and the "bound" (or shape) b is an entity that can be interpreted as a predicate $[\![b]\!]$. (f, b) can be interpreted as a partial function where b provides a safe approximation of its domain. The canonical example is the array: then b is a tuple of index bounds and $[\![b]\!]$ is a conjunction of linear inequalitites which returns *true* for indices within the bounds and *false* otherwise. Data fields are, however, designed in an abstract fashion which makes it possible to encompass many other structures than arrays. The design of data fields and operations on them is guided by their interpretation as partial functions. Thus, before providing a more exact definition of data fields and their operations we will develop the more abstract model of indexed data structures as partial functions.

The term "data fields" is borrowed from Crystal [4].

2 Partial Functions

We use the following representation of partial functions. "Calls" to a partial function falling "out of bounds" return a distinguished *error value* $*$. Semantically this is a total, isolated element. It is thus distinct from the bottom element $\perp$ although it is supposed to have similar algebraic properties, i.e., strict functions are supposed to be strict in $*$ as well (some complications will however arise for operations strict in more than one argument [15]).

For any function $f\colon D \to D'$ where $* \in D'$ we define the *index set* of f as the set of all $x \in D$ such that x is a total, compact element and $f(x) \neq \perp$. This is supposed to model the set of indices where an indexed data structure is defined.

In the following we will use a simple metalanguage for partial functions defined as above, consisting of λ-abstraction, simple n-ary operations "*op*" strict in all arguments, a conditional "*if*", the constant $*$, and possibly other constructs added as the need arises. If we add recursion we have a PCF-like language [16]. The conditional is defined as usual with the addition that $\mathit{if}(*, x, y) = *$ (since *if* is strict in its first argument).

2.1 Operations on Partial Functions

All the five kinds of operations on indexed data structures described in Sect. 1.1 can be expressed as higher order operations on partial functions:

- *Elementwise application* of an operation g on the partial functions $f_1, \dots, f_n$ is a kind of function composition: $\lambda x.g(f_1(x), \dots, f_n(x))$. It will often be convenient to use "elemental intrinsics style overloading" of g and write $g(f_1, \dots, f_n)$. We will make frequent use of this syntax.

- *Parallel read* of the partial function f w.r.t. source function g is also function composition, but to the "right": $\lambda x.f(g(x))$ (or $f(g)$). Parallel write is not so straightforward to model, due to its nondeterministic nature, but certain deterministic variants can be defined with the aid of reduction [9, 15].

- *Replication* is λ-abstraction with respect to a fresh variable. If x does not occur free in t, then $\lambda x.t$ is independent of x and can be seen as the value of t replicated to each possible index value for x.

- *Explicit restriction* of a partial function f w.r.t. the predicate b is defined viz.: $f \setminus b = \lambda x.\mathit{if}(b(x), f(x), *)$.

- *Reduction*, finally, can be expressed through a simple recursion, if the *size* of the index set as well as an *enumeration* of this set is provided [9, 15].

It should now be fairly obvious how the three syntactical styles for operations on indexed data structures in Sect. 1.1 appear in our metalanguage for partial functions. The first kind of syntax is obtained by explicitly defining higher order functions, e.g., $\mathit{funplus}(f,g) = \lambda x.(f(x) + g(x))$. The elemental intrinsics style of syntax is obtained as overloading of application resolved into function composition. The third, "quantified" syntax for operations, finally, exemplified by the `FORALL`-construct in Fortran 90 and HPF, is simply given by lambda abstraction.

2.2 Laws for Explicit Restriction

A number of algebraic laws can be proved for explicit restriction. They are interesting because they can be seen as propagation laws for predicates restricting partial functions. Restricting predicates for partial functions correspond to bounds for data fields. Thus, the laws for explicit restriction can guide the design of operations on data fields, e.g., how bounds for data fields resulting from elementwise applied operations should be calculated from the bounds of the arguments.

In the following we use elementwise intrinsics syntax throughout. "$\wedge$" and "$\vee$" refer to the non-strict versions of conjunction and disjunction, respectively, for which $\mathit{false} \wedge \perp = \mathit{false}$ and $\mathit{true} \vee \perp = \mathit{true}$, extended to handle $*$ in the following way:

$$\begin{array}{rcl@{\qquad}rcl}
* \wedge x & = & * & * \vee x & = & * \\
\mathit{true} \wedge * & = & * & \mathit{true} \vee * & = & \mathit{true} \\
\mathit{false} \wedge * & = & \mathit{false} & \mathit{false} \vee * & = & * \\
\perp \wedge * & = & \perp & \perp \vee * & = & \perp
\end{array}$$

Flattening of nested restrictions:

$$(f \setminus b) \setminus b' = f \setminus (b' \wedge b) \tag{1}$$

Communication of restriction:

$$(f \setminus b)(g) = f(g) \setminus b(g), \tag{2}$$

and if g has a left inverse g^{-1}, then

$$f(g) \setminus b = (f \setminus b(g^{-1}))(g) \tag{3}$$

Assume that g is strict in all its arguments. The following law holds under the condition that $*$ *and* $\perp$ *are identified*, i.e., that the equation $\perp = *$ is valid.

$$g(f_1 \setminus b_1, \ldots, f_n \setminus b_n) = g(f_1, \ldots, f_n) \setminus (b_1 \wedge \ldots \wedge b_n) \tag{4}$$

It is possible to find conditions under which (4) holds also when $*$ and $\perp$ are distinguished [15]. Finally, for the elementwise applied conditional, we have:

$$\mathit{if}(b, f \setminus b_1, g \setminus b_2) = \mathit{if}(b, f, g) \setminus (b \wedge b_1) \vee ((\neg b) \wedge b_2) \tag{5}$$

3 Data Fields

For certain domains α we assume the existence of a domain $\mathcal{B}(\alpha)$ of *bounds* for α. A simple example is $\mathcal{B}(\mathit{int}) = \mathit{int} \times \mathit{int}$, i.e., pairs of lower and upper bounds for one-dimensional arrays. For any domains α, β, where $\mathcal{B}(\alpha)$ is defined, the domain of *data fields from* α *to* β, $\mathcal{D}(\alpha, \beta)$, is defined by

$$\mathcal{D}(\alpha, \beta) \cong (\alpha \to \beta) \times \mathcal{B}(\alpha)$$

In the sequel, symbols f range over $\alpha \to \beta$, d over $\mathcal{D}(\alpha, \beta)$, and b over $\mathcal{B}(\alpha)$. Thus, a data field is a pair (f, b). We assume that the following holds for $\mathcal{B}(\alpha)$:

- $\mathcal{B}(\alpha) = \mathcal{B}^{\infty}(\alpha) + \mathcal{B}^{\mathit{fin}}(\alpha)$, where $\mathcal{B}^{\infty}(\alpha)$ is a domain of *infinite bounds* for α and $\mathcal{B}^{\mathit{fin}}(\alpha)$ is a domain of *finite bounds*.
- Every bound $b \in \mathcal{B}(\alpha)$ has an interpretation as a predicate $[\![b]\!] \colon \alpha \to \mathit{bool}$.
- There is an operation "!" for *data field application* defined by $d \,!\, x = [\![d]\!]x$ for all $d \in \mathcal{D}(\alpha,\beta)$ and $x \in \alpha$.
- Every data field $(f,b) \in \mathcal{D}(\alpha,\beta)$ has an interpretation $[\![(f,b)]\!] = \overline{f \setminus [\![b]\!]}$ in $\alpha \to \beta$. ($\overline{f}$ denotes the hyperstrict version of f, for which $\overline{f}(x) = f(x)$ when x is a maximal, compact element and $\overline{f}(x) = \bot$ otherwise.)
- For every finite bound $b \in \mathcal{B}^{\mathit{fin}}(\alpha)$ there are total, hyperstrict functions $\mathit{enum} \colon \mathcal{B}(\alpha) \to (\mathit{int} \to \alpha)$ and $\mathit{size} \colon \mathcal{B}(\alpha) \to \mathit{int}$ which *enumerate* the set defined by a bound and give the *size* of this set, respectively. We require that $\mathit{size}(b) \geq 0$. If $\mathit{size}(b) = 0$ then b is *empty*, otherwise *nonempty*. We require the existence of an $\mathit{enum}(b)$ with the properties above only when b is nonempty.
- There are binary operations $\sqcap$, $\sqcup$ on $\mathcal{B}(\alpha)$. These are supposed to correspond to elementwise $\wedge$ and $\vee$ on predicates.
- $\mathcal{B}(\alpha)$ contains a particular element *all* such that $[\![\mathit{all}]\!] = \lambda x.\mathit{true}$, and $\mathcal{B}^{\mathit{fin}}(\alpha)$ contains an element *nothing* such that $[\![\mathit{nothing}]\!] = \lambda x.\mathit{false}$ and $\mathit{size}(\mathit{nothing}) = 0$.
- If $\mathcal{B}(\alpha_1), \ldots, \mathcal{B}(\alpha_k)$ are defined (for $k > 1$), then $\mathcal{B}(\alpha_1 \times \cdots \times \alpha_k)$ is defined, and there is an operation $\times_k \colon \mathcal{B}(\alpha_1) \times \cdots \times \mathcal{B}(\alpha_k) \to \mathcal{B}(\alpha_1 \times \cdots \times \alpha_k)$ for forming k-dimensional bounds such that:

$$\begin{aligned}
[\![\times_k(b_1,\ldots,b_k)]\!](x_1,\ldots,x_k) &= [\![b_1]\!](x_1) \wedge \cdots \wedge [\![b_k]\!](x_k) && (6)\\
\mathit{size}(\times_k(b_1,\ldots,b_k)) &= \mathit{size}(b_1) \cdot \ldots \cdot \mathit{size}(b_k) && (7)\\
\mathit{enum}(\times_k(b_1,\ldots,b_k),n) &= (\mathit{enum}(b_1)(n \bmod \mathit{size}(b_1)),\\
&\quad\ \mathit{enum}(\times_{k-1}(b_2,\ldots,b_k))(n \div \mathit{size}(b_1))),\\
&\quad\quad \text{when } \mathit{size}(b_1) > 0 && (8)
\end{aligned}$$

  for all $b_i \in \mathcal{B}(\alpha_i)$, $1 \leq i \leq k$ ($b_i \in \mathcal{B}^{\mathit{fin}}(\alpha_i)$ for (7) and (8)), and $n \in \mathit{int}$. We define $\times_1(b) = b$. We do not specify the strictness properties of $[\![\times_k(b_1,\ldots,b_k)]\!]$ fully. "$\div$" is integer division.

It is straightforward to verify that $\mathit{size}(\times_k(b_1,\ldots,b_k))$ and $\mathit{enum}(\times_k(b_1,\ldots,b_k))$ fulfil the requirements for *size* and *enum* functions, provided that *size* and *enum* for $b_1,\ldots,b_k$, respectively, do.

Inspired by (1), we define *explicit restriction* of data fields, $\downarrow \colon \mathcal{D}(\alpha,\beta) \times \mathcal{B}(\alpha) \to \mathcal{D}(\alpha,\beta)$, by $(f,b) \downarrow b' = (f, b' \sqcap b)$ for all data fields (f,b) and bounds b.

3.1 Data Field Evaluators and Reductions

A function is evaluated only when applied to an argument. The major difference between data fields and partial functions is that we want a mechanism to evaluate all the possibly defined elements of a data field. This mechanism could be seen as a hyperstrict environment requesting all these elements. It would be called by a print function, just as for arrays and lists occurring at the top level of an interpreter for a functional language. The *data field evaluator* $[\,\cdot\,] : \mathcal{D}(\alpha,\beta) \to \mathcal{D}(\alpha,\beta)$, defined by the following equations, is such a mechanism:

$$\begin{array}{rcl} [(f,b)] & = & \overline{(\lambda x.\mathit{lookup}(x, \overline{\mathit{genlist}(0,f,b)}, *), b)}, \\ & & \text{when } b \in \mathcal{B}^{\mathit{fin}}(\alpha) \\ \mathit{genlist}(n,f,b) & = & \mathit{if}(n = \mathit{size}(b), \mathit{NIL}, \\ & & \quad (\mathit{enum}(b,n), f(\mathit{enum}(b,n))){:}\mathit{genlist}(n+1,f,b)) \end{array}$$

Thus, $[\,\cdot\,]$ is an operator that takes a basic data field (f,b), examines b, enumerates the index set defined by $[\![b]\!]$ if b is a finite bound, creates a table of all defined elements of (f,b), and returns a new data field where the body is a lookup function into the table. The central part of the evaluator is the call to *genlist* which produces an association list of the values. A print function would call *genlist* directly. Given the *enum* and *size* functions for bounds, it is also easy to define reduction operations for data fields.

3.2 φ-abstraction

φ-abstraction is a syntax with bound variables for data fields (a functional, generalized `FORALL`): If x has type α and t has type β, then $\varphi x.t$ has type $\mathcal{D}(\alpha,\beta)$. φ-abstraction parallels λ-abstraction for functions: the idea is that φ-abstraction should provide a syntax for defining data fields which is as simple and general as λ-abstraction is for defining partial functions as shown in Sect. 2.1. We define φ-abstraction to propagate bounds similar to the laws for explicit restriction in Sect. 2.2.

The semantics of φ-abstraction over a given host language is given as a higher order rewrite system $\Phi(R)$ (a Combinatory Reduction System, or CRS [11, 12]). R is a CRS which gives semantics for the host language in question. $\Phi(R)$ consists of a number of reduction rules of the form

$$\varphi\vec{x}.t \to (\lambda\vec{x}.t, B(t,\vec{x},\emptyset))$$

for all t such that $B(t,\vec{x},\emptyset)$ is defined. $\vec{x}$ denotes the tuple $(x_1,\ldots,x_n)$: thus, $\varphi\vec{x}.t = \varphi(x_1,\ldots,x_n).t$. Furthermore, $B(t,\vec{x},\emptyset) \in \mathcal{B}(\vec{\alpha})$ when $\vec{x}$ has type $\vec{\alpha}$ $(= \alpha_1 \times \cdots \times \alpha_n)$. In general, B is a function which takes terms, tuples of variables, and sets of variables into bounds. B is a partial function, and we require that $B(t,\vec{x},Y)$ should be defined if and only if:

- $FV(t) \subseteq \vec{x} \cup Y$,
- t is a R-nf, and
- t has no closed subterm of the form $\varphi\vec{y}.t'$.

Here and henceforth, $\vec{x}$ also stands for $\{x_1,\ldots,x_n\}$ when implied by the context.

It is easy to prove that $\Phi(R)$ is orthogonal and left-normal. Furthermore, if R is left-linear and if no left-hand sides of any rules in R have subterms of the form $\varphi\vec{x}.t$, then $\Phi(R)$ and R are mutually orthogonal. If, in addition, R is orthogonal and left-normal, then $\Phi(R) + R$ is orthogonal and left-normal.

$B(t,\vec{x},Y)$ is defined as follows, for different forms of t, under the condition that t is such that $B(t,\vec{x},Y)$ should be defined according to above. In cases not covered, where $B(t,\vec{x},Y)$ still should be defined, we assume a default definition

$$B(t,\vec{x},Y) = \mathit{all}.$$

First some straightforward cases:

$$\begin{aligned}
B(c,\vec{x},Y) &= \mathit{all} \qquad (c \text{ constant}) && (9)\\
B(x,\vec{x},Y) &= \mathit{all} \qquad x \in \vec{x} \cup Y && (10)\\
B(op(t_1,\dots,t_m),\vec{x},Y) &= B(t_1,\vec{x},Y) \sqcap \cdots \sqcap B(t_m,\vec{x},Y) \quad (op \text{ strict}) && (11)\\
B(\mathit{if}(t_1,t_2,t_3),\vec{x},Y) &= B(t_1,\vec{x},Y) \sqcap (B(t_2,\vec{x},Y) \sqcup B(t_3,\vec{x},Y)) && (12)\\
B(\lambda\vec{y}.t,\vec{x},Y) &= B(t,\vec{x},\vec{y}\cup Y) && (13)\\
B(\varphi\vec{y}.t,\vec{x},Y) &= B(t,\vec{x},\vec{y}\cup Y) && (14)
\end{aligned}$$

They are motivated by the desired similarity between the index set of the partial function $\lambda\vec{x}.t$ and the bounds of the data field $\varphi\vec{x}.t$. In particular, (11) is motivated by (4) and (12) by (5).

Then comes the interesting case where a data field is applied to some variable(s) in $\vec{x}$. These cases will have the form

$$B((f,\times_n(b_1,\dots,b_n))\,!\,F(\vec{x}),\vec{x},Y) = \dots$$

A minimalistic approach would be to consider only the one-dimensional case, with F as the identity function, and define

$$B((f,b)\,!\,x,x,Y) = b.$$

This is probably satisfactory if we do not pay particular attention to multidimensional data fields. We however make a quite general and complex definition, subsuming the case above, which covers the case when F maps $\vec{x}$ to an argument containing elements of $\vec{x}$ mapped in some fashion, possibly interspersed with constants. The complexity is motivated by the need to perform operations on array- and array-like data like transposition, selection of a row or column, etc., which are covered by our definition. The details are found in [15].

As an example, consider $B((f,b)\,!\,(c,x_3,x_2,x_2),(x_1,x_2,x_3),\emptyset)$, where $b = \times_4(b_1,b_2,b_3,b_4)$. Intuitively, this bound should be *nothing* if c falls outside the set defined by b_1, otherwise x_3 should be constrained by b_2, x_2 by $b_3 \sqcap b_4$, and x_1 should be unconstrained. That is, the bound should equal $\mathit{if}([\![b_1]\!]\,c,\times_3(\mathit{all},b_3\sqcap b_4,b_2),\mathit{nothing})$. A resulting rule for reduction of φ-abstraction is

$$\begin{gathered}
\varphi(x_1,x_2,x_3).(f,b)\,!\,(c,x_3,x_2,x_2) \to\\
(\lambda(x_1,x_2,x_3).(f,b)\,!\,(c,x_3,x_2,x_2),\mathit{if}([\![b_1]\!]\,c,\times_3(\mathit{all},b_3\sqcap b_4,b_2),\mathit{nothing})).
\end{gathered}$$

Other interesting examples are

$$\varphi(x_1,x_2).(f,\times_2(b_1,b_2))\,!\,(x_2,x_1) \to (\lambda(x_1,x_2).(f,\times_2(b_1,b_2))\,!\,(x_2,x_1),\times_2(b_2,b_1))$$

(transposition) and

$$\varphi x.(f,\times_2(b_1,b_2))\,!\,(x,x) \to (\lambda x.(f,\times_2(b_1,b_2))\,!\,(x,x),b_1\sqcap b_2)$$

(selection of diagonal).

We believe that φ-abstraction provides an interesting format for generic definitions of data fields, since the rules in $\Phi(R)$ are expressed with the general bounds operators $\sqcup$, $\sqcap$, $\times$ and the polymorphic bounds constants *all* and *nothing*. Syntactical conveniences can be given a semantics in terms of φ-expressions. For instance, elemental intrinsics style overloading can be resolved into such expressions:

$$\begin{aligned}
X+Y &\Rightarrow \varphi x.(X\,!\,x+Y\,!\,x)\\
\mathit{if}(A<0,-A,A) &\Rightarrow \varphi x.\mathit{if}(A\,!\,x<0,-A\,!\,x,A\,!\,x)
\end{aligned}$$

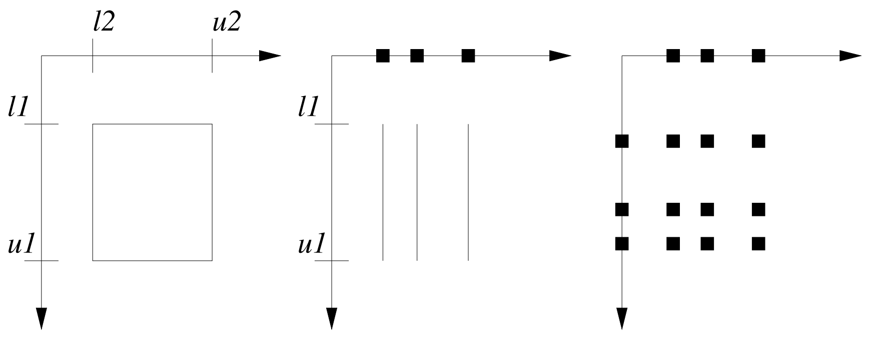

Figure 1: Some bounds in $\mathcal{B}_{arr}(int^2)$, of type: $(int \times int)^2$, $(int \times int) \times List\ int$, $(List\ int) \times (List\ int)$, and $List\ int^2$.

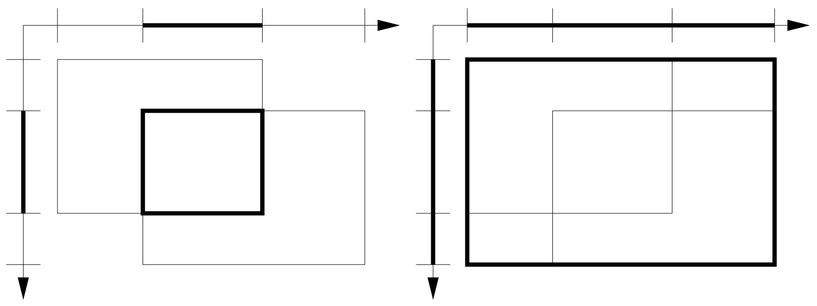

Figure 2: $\sqcap$ and $\sqcup$ for array bounds $(l_1, u_1) \times (l_2, u_2)$, $(l'_1, u'_1) \times (l'_2, u'_2)$.

Note that these φ-expressions will reduce in a uniform way through rules in $\Phi(R)$ regardless of the "shapes" of X, Y and A: the shape of $X + Y$ will be the "intersection" ($\sqcap$) of the shapes of X and Y, and $if(A < 0.0, -A, A)$ will have the same shape as A regardless of whether it is array-like, or sparse, or whatever. Some more developed examples are given in [15].

3.3 Bounds for Sparse and Dense Arrays

As an example how the data fields can go beyond traditional arrays, we define domains of *array bounds* $\mathcal{B}_{arr}(int^n)$ for array-like mixed sparse-dense data fields:

$$\mathcal{B}^{fin}_{arr}(int) = (int \times int) + List\ int + Nothing \tag{15}$$

$$\mathcal{B}^{\infty}_{arr}(int) = All + (int \to bool) \tag{16}$$

$$\mathcal{B}_{arr}(int) = \mathcal{B}^{\infty}_{arr}(int) + \mathcal{B}^{fin}_{arr}(int) \tag{17}$$

$$\mathcal{B}_{arr}(int^n) = \mathcal{B}_{arr}(int)^n + All + (int^n \to bool), \quad n > 1 \tag{18}$$

All is the two-point cpo with non-bottom element *all*, and *Nothing* the one with non-bottom element *nothing*. $int \times int$ is the cpo of one-dimensional dense array bounds, where each integer pair defines an array range. *List int* is the domain of finite, sorted lists of integers. It provides *sparse* bounds for array coordinates. Some examples of bounds in $\mathcal{B}_{arr}(int^2)$ are shown in Fig. 1.

$\mathcal{B}^{fin}_{arr}(int^n)$ equals $\mathcal{B}^{fin}_{arr}(int)^n$ (i.e., an n-dimensional bound is finite iff it is a product of finite one-dimensional bounds). All other bounds in $\mathcal{B}_{arr}(int^n)$ are infinite.

Exact definitions for $[\![\ ]\!]$, *size*, *enum*, $\sqcap$ and $\sqcup$ are given in [15]. For a simple case, see Fig 2.

4 Ongoing Work

Our work so far has mainly consisted of the formulation of abstract models. Besides a small early implementation experiment [8], little implementation work has been done. We are currently investigating the possibility to extend Haskell with data fields. This choice of host language is motivated by the possibility to do much of the rapid prototyping of data fields in Haskell itself, by the strong typing, which seems essential if syntactical conveniences such as elemental intrinsic style overloading is to be included, and also by the class system which seems apt for bounds. Haskell's class system would also make it possible to add user-defined bounds, tailored to specific classes of applications.

The introduction of elemental intrinsics overloading in a language with type inference á la Hindley-Milner requires a modification in the type system to make the overloading and type inference coexist without interference. In Haskell, this overloading could be defined on a per-operator basis but it would be nice to have it automatically available for all functions, including user-defined ones. We are currently working on a modified Hindley-Milner type system which provides this [18].

References

[1] Eugene Albert, Joan D. Lukas, and Guy L. Steele Jr. Data parallel computers and the `forall` statement. *J. Parallel Distrib. Comput.*, 13:185–192, October 1991.

[2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, April 1994.

[3] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to FORTRAN 90.* Programming Languages. McGraw-Hill, 1990.

[4] Marina C. Chen, Young-Il Choo, and Jingke Li. Crystal: Theory and pragmatics of generating efficient parallel code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7, pages 255–308. 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] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1):1–170, June 1993. HPF Version 1.0.

[8] Joacim Halén, Per Hammarlund, and Björn Lisper. An experimental implementation of a highly abstract model of data parallel programming. Technical Report TRITA-IT R 97:02, Dept. of Teleinformatics, KTH, Stockholm, March 1997.

[9] 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.

[10] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, December 1986.

[11] Jan Willem Klop. *Combinatory Reduction Systems.* PhD thesis, CWI, Amsterdam, 1980. Mathematical Centre Tracts Nr. 127.

[12] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoret. Comput. Sci.*, 121:279–308, 1993.

[13] 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.

[14] Björn Lisper and Jean-François Collard. Extent analysis of data fields. In Baudouin Le Charlier, editor, *Proc. International Symposium on Static Analysis*, Vol. 864 of *Lecture Notes in Comput. Sci.*, pages 208–222, Namur, Belgium, September 1994. Springer-Verlag.

[15] Björn Lisper and Per Hammarlund. The data field model. Submitted., 1998.

[16] Gordon Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5(3):223–256, December 1977.

[17] Stephen K. Skedzielewski. Sisal. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 4, pages 105–157. Addison-Wesley, 1991.

[18] Claes Thornberg. Polymorphic type inference with elemental function overloading. Licentiate Proposal, June 1997.