Data Parallelism and Functional Programming

Björn Lisper

Department of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, SWEDEN

Abstract. *Data parallelism* is often seen as a form of explicit parallelism for SIMD and vector machines, and *data parallel programming* as an explicit programming paradigm for these architectures. Data parallel languages possess certain software qualities as well, which justifies their use in higher level programming and specification closer to the algorithm domain. Thus, it is interesting to study how the data parallel paradigm can be best realized in a declarative setting, since declarative languages offer a pure view of computation which is good for these purposes. For numerical computing the functional programming paradigm is especially attractive, since numerical algorithms often are specified by recursion equations and thus can be translated more or less directly into recursive functional programs. Merging the data parallel and functional paradigms then yields languages and formalisms where many algorithms can be expressed in a very succinct fashion. In this paper we review data parallelism, functional programming, and existing approaches to the integration of the two paradigms. We then proceed to describe a formalism for data parallel functional programming, allowing very simple languages, where the view of aggregate data is particularly abstract. We explain how various data parallel operations can be expressed in this formalism. Finally, we conclude with a discussion of issues for languages based directly on the formalism.

1 Introduction

This paper gives a review of a novel abstract programming formalism [28, 29]. The source of inspiration is the data parallel paradigm and its view of aggregate data, which often makes it possible to write very clear and succinct data parallel programs for certain applications. The main objective has been to *generalize* current data parallel constructs, thereby making the data parallel programming style applicable to a wider class of applications, and to give a more *abstract* view of aggregate data, thereby making the programming model less tied to certain choices of data representations and hardware architectures. The motivation for this is the observation that different existing data parallel languages and formalisms use different carriers of parallel data, yet they clearly share a common set of concepts. Our desire has been to find a formalism where this "essence of data parallelism" is present in its most pure form.

Another objective has been to support a *simple* programming model, both with respect to *language constructs* (a small number of primitives should suffice) and *semantics*, to pave the way for languages which are easy to understand and

use. Furthermore, we would like these languages to have all the good features of modern programming languages which make it possible to write safe, modular, and reusable code.

Apparently, it is then interesting to study how data parallelism can be integrated with declarative languages, since these languages offer the most abstract and semantically clean programming models. In particular, the *functional* paradigm is interesting, since pure functional programs have a very simple abstract semantics, since they can provide excellent support for modularity, reusability and safety, and since it turns out that the abstract formulations of many computational algorithms are very close to functional programs.

It turns out that an abstract formalism for aggregate data can be based directly on the functional paradigm. The result is a very simple formalism for aggregate data which is based on the view that data structures really represent partial functions. The formalism has at least three different uses:

- to give semantics to existing parallel data types and operations on them,
- as a formal framework where certain *algebraic laws*, which can be used to transform programs and data, can be proved,
- as a direct basis for novel data parallel programming languages.

We will treat all these three aspects in this paper. But let us first give some background:

2 Data Parallelism

Data parallelism usually refers to the kind of parallelism where operations are carried out in parallel over collections of data: see also [10] for a similar discussion of the topic. This kind of parallelism is conceptually simple and *deterministic* (i.e., computing with the same inputs will always yield the same result), in contrast to process parallelism where asynchronous updates of common data can give rise to nondeterminism (the result of a computation will depend also on factors outside the program control, such as relative timing between processes.)

The data parallel paradigm first arose as a form of explicit parallelism for SIMD machines. Some data parallel production languages for SIMD machines, e.g., C* and *Lisp for the Connection Machine [54, 55], reflect the underlying architecture quite closely. It was, however, soon discovered that data parallelism also possesses certain software qualities: the use of operations on aggregate data can often yield very succinct programs for applications where such data are extensively handled. This makes data parallel programming very apt for large scale computing applications, where large data sets are handled in an often quite uniform fashion. The deterministic nature of data parallelism is another advantage, since deterministic programs are easier to understand and debug but also since the algorithms being used in computing applications most often are deterministic. Therefore, data parallel languages make it possible to program closer to the algorithm domain.

What are data parallel operations like, then? They can roughly be classified in five groups.

2.1 Elementwise Applied Operations

These data parallel operations apply a "scalar" operation f to every element a in a data collection A. That is, the resulting data collection will consist of the elements $f(a)$, where a belongs to A. See Fig. 1.

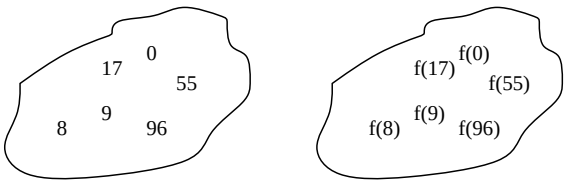

Fig. 1. A elementwise applied operation f.

Fig. 1 suggests that data parallel entities could be seen as sets, or multisets (to allow several occurrences of the same datum). For unary operations f being elementwise applied, this view can indeed be adequate. However, one often wants to apply operations, which take *several* arguments, elementwise, like *adding* two data collections elementwise. The operational view is then that the data collections all are distributed over some parallel machine, with one element each per processor, and each processor will apply the scalar operation to its respective local elements of the data collections. For instance, if two data collections A and B are to be elementwise added, then each processor will add its local elements of A and B. This implies that data collections must be seen as *indexed* structures, where the indexing indicates how individual elements are to be combined. Unstructured collections such as sets or multisets do not provide the information necessary to do this. Thus, we will from now on consider data collections to be indexed over some (more or less abstract) set of locations, and we let $A(i)$ denote the element residing at the location i of the data collection A. From now on, in analogy with the terminology in physics, we will refer to these indexed collections as *data fields* (as in the language Crystal [17]). The *index set* of a data field is the set of locations where it is defined.

What about program notation for elementwise applied operations? There are several possibilities. The notation closest to an actual implementation is to assume a syntactically distinct parallel operation for each "scalar" operation being elementwise applied. For instance, if p_add adds two data fields elementwise, then

$$C := p\_add(A, B)$$

assigns the elementwise sum of the data fields A and B to C. This kind of operation is used in *Lisp [55] for the Connection Machine, where elementwise addition is denoted "`+!!`". It is often more convenient to *overload* the scalar operation, viz.:

$$C := A + B$$

Array languages like Fortran 90 [11] provide this kind of overloading. This assumes some kind of typing of A and B, such that the overloading really can be resolved. This kind of overloaded syntax is also common in mathematical texts on, e.g., linear algebra, and should thus be familiar to many programmers.

A third form of notation "quantifies" over the index set, in order to explicitly mention each of the individual, "scalar", elements of the data fields involved:

$Forall\ i\ \ C(i) := A(i) + B(i)$

Here, as well as in he other two cases, the index set is implicitly given and i ranges over that set. It is common that i is given an explicit range, e.g., if the locations are integers, "$Forall\ i\ in\ 1..N\ \ \ldots$". High Performance Fortran (HPF) [30] provides `FORALL` statements with range.

2.2 Communication Operations

The second main group of data parallel operations concern communication. *Get communication*, or *parallel* (concurrent) *read*, lets each location i concurrently read an element of some data field A at location $source(i)$. Here, $source$ is some total function from locations to locations. See Fig. 2 for an example, where $source(a) = c$, $source(b) = a$, $source(c) = b$, $source(d) = d$, $source(e) = d$, and $source(f) = d$.

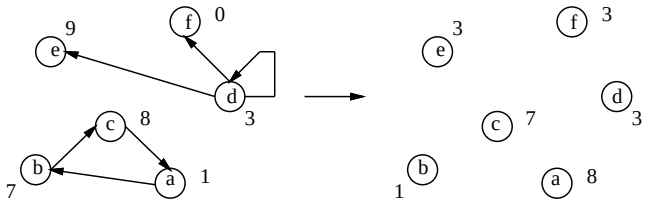

Fig. 2. Get communication. Each location has exactly one source.

Get communication can be expressed in the same three ways as above: using an explicit operation, say, p_get,

$B := p\_get(A, source)$,

or, with overloaded syntax,

$B := A(source)$,

or, with "$Forall$",

$Forall\ i\ \ B(i) := A(source(i))$.

The other main communication operation is *send communication*, where each location i concurrently sends its element of the data field A to the location $dest(i)$, where it possibly is received into the local element of the data field B.

Send communication can thus be seen as having side effect, and is then really a special case of concurrent write in the PRAM model of parallel computing. See Fig 3. The syntax, assuming a procedure *p_send* modifying its first argument, can be:

$p\_send(A, dest, B)$,

with overloaded syntax:

$B(dest) := A$,

and, with "*Forall*",

$Forall\ i\ \ B(dest(i)) := A(i)$.

If $dest(i) = dest(j)$ for some distinct locations i, j, then there is a write conflict. As in the CRCW PRAM model, this conflict can be resolved in various ways, e.g., let the result be nondeterministically chosen among the values, abort with error, or in some way *combine* (typically *reduce*, see Section 2.5) the elements being sent to the same address. For instance, if the elements are integers, then they can be added. This is sometimes referred to as *combining send*.

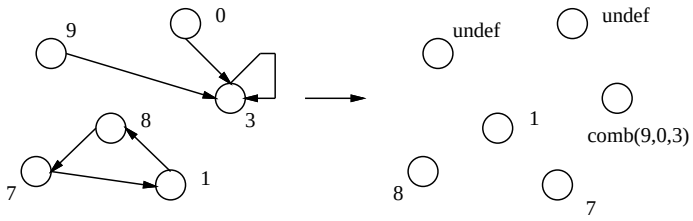

Fig. 3. Send communication. Each location has exactly one destination. Certain locations may not be targets of any sending locations (receiving "undef"): for send with side effect, the destination data field is not touched in these locations.

2.3 Replication

It is common to have shorthand notations for data fields filled with copies of a single scalar. For instance, the functional array language Sisal [24] has an operation `array_fill` that creates an array where each element is a copy of a given scalar. In languages with overloading of elementwise applied scalar operations, the following syntax is often allowed:

$C := A + 17$

If A is a data field, then each $C(i)$ is assigned $A(i)+17$. This can be seen as a two-step operation where first a data field with the same index set as A, filled with the scalar 17, is created, and then these data fields are elementwise added. This automatic replication of a scalar into a data field is sometimes called *promotion*.

Replication can be seen as *broadcast communication*: if the resulting data field is distributed among a number of processors, then the scalar must indeed be broadcast to them.

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.

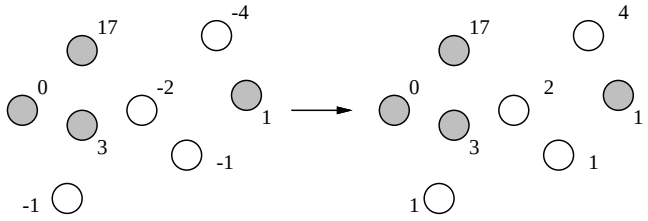

Fig. 4. Restriction to mask out locations with negative value in order to change their sign.

2.4 Restriction

This kind of operation applies a boolean data field elementwise, as a "mask", in order to restrict the number of "active" locations. The restriction operation creates a local environment, where some locations are "turned off", and data parallel operations within this environment take place only in the active locations. A possible syntax is "*Where mask do statement*", e.g., (using overloaded syntax),

$$\mathit{Where}\ A < 0\ \ \mathit{do}\ A := -A$$

which effectively sets every element of A to its absolute value. Fig. 4 provides an illustration.

2.5 Reductions

This is a group of operations which compute some scalar value as a function of the elements of a data field. Thus, they in general involve both communication and arithmetics, since information must be gathered from different locations. Usually, the function is composed of some repeatedly applied binary operation. An example of a reduction is computing the sum of all elements in a data field, which is an operation formed from repeated addition: see Fig. 5.

If the binary operation is associative, then the reduction can be performed according to a balanced binary expression tree, with height logarithmic in the number of scalar operations performed. Unless the operation is also commutative, the reduction needs to be performed according to a fixed *order* on the

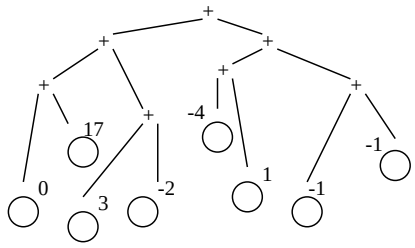

Fig. 5. Summing all elements in a data field.

locations in order to be well-defined. This order can be given either implicitly, e.g., by the less-than order if the locations are integers, or explicitly as an argument to the reduction, e.g., as a function from integers to locations enumerating the locations. The former case is by far the most frequent.

Very related are the *scan* (or *parallel prefix*) operations [32]. They compute all partial reductions over a data field with respect to some binary operation. The result is a data field over the same index set as the original field. For instance, if A is a data field indexed over $1..n$, then, after executing $B := scan(A, +)$, we should have $B(i) = \sum_{j=1}^{i} A(j)$ for all i between 1 and n.

The scan operation requires an ordering on the locations in order to be well-defined, even when the binary operation is associative and commutative.

2.6 Index Sets

Machine-oriented data parallel languages have data fields defined over a global index set, which then invariably have the processor addresses of the machine as locations. This makes it simple for a compiler to distribute data fields onto the machine, and the language can be used as a means for accurate low-level programming for that machine. A global index set also makes it possible, in the presence of a restriction operation, to have a global state of "currently active locations". This state can often be implemented very directly on SIMD machines as a boolean mask of local context flags deciding, for each processor, whether it should participate in the operations or not.

Being confined to a global global index set can be very restraining, however. Rather, one would like to associate an individual index set with every data field. This is the model supported by array languages like Fortran 90 and HPF. It makes it possible to have coexisting data fields of different type (e.g., vectors, matrices) or of the same type but with different ranges. Thus, algorithms dealing with such entitites can be modeled more closely. However, some semantical complications now arise:

- For restrictions, the mask must range over the locations of the data fields used inside the restriction. For instance, if A is an array and B is a matrix, then it does *not* make sense to write *Where* $A < 0$ *do* $B := -B$.

– The semantics of certain operations can become unclear. Consider, for instance, the elementwise addition $C := A + B$ of two arrays A, B. The ranges of A and B may now well be different, say, $1..n-1$ for A and $1..n$ for B (they are *non-conformant* in the Fortran 90 terminology, since they have different length). What, then, is the range of C, $1..n-1$ or $1..n$? What should the value of $C(n)$ be, if defined? Should A be padded with a zero to give it the same range as B before adding them? What, then, if we instead of adding the arrays want to apply a binary operation elementwise which has no identity element? Should, perhaps, addition of non-conformant arrays like A and B be disallowed and result in a runtime error? If we, instead, change the range of B to $2..n$, should the arrays now be considered conformant, as in Fortran 90, and an implicit shift of B take place before performing the addition? Or should we rather take the "data parallel" view, where really $C(i) = A(i) + B(i)$ when defined?

Resolving these semantical issues in a natural and general way was one of the initial goals of our work. This led to the abstract model of data fields presented in Section 5.

3 Functional Languages

Functional languages [34] are a class of declarative languages usually considered to have the following properties:

– No side effects, no destructive updates.
– Program control is only through nonstrict operators (see below) and recursion.
– A program is a function definition (possibly recursive).
– Computations are triggered by demanding the result of evaluating some expression (which may contain calls to user-defined functions).

Another way to formulate the "no side effect" condition is that functional languages hide the handling of memory completely, whereas imperative languages, through side effects, give the programmer explicit control over where and when a certain value is to reside in memory. Thus, a functional program is typically less specific about the ordering of computations and is usually easier to parallelize than its imperative counterpart, since there are no memory-carried dependences. For this reason, functional programming has been put forward as a suitable paradigm for implicit parallel programming. On the other hand, the memory handling can be a major source of inefficiency, as well as the more sophisticated mechanisms necessary to handle recursion as compared to iteration. These effects can well negate the gains from the increased parallelism. (It is interesting to note, though, that advanced parallelizing compilers for imperative programs do use the essentially functional *single-assignment format* internally in order to enhance the parallelization. Thus, it seems like there is some common ground to be explored between parallelizing compilers for functional and imperative programs.)

However, functional languages have other interesting properties. They are *deterministic*, since function definitions really do implement functions in the mathematical sense. It is often simple to define strong *type systems* for functional languages. These systems make it easier to write correct programs since they provide a kind of compile-time error checking w.r.t. the proper use of data, and they can also make it possible to generate more efficient code due to the type information. Finally, functional languages can be given a simple, abstract *semantics*. This makes it easier to reason about programs and understand them.

Type systems for functional languages normally have *function types*, i.e., if α, β are types, then $\alpha \to \beta$ is the type of functions which take an argument of type α and return a value of type β. A natural step is to allow α and β to be function types themselves. We then obtain a *higher order functional language*, where functions are considered values and can be passed around just like any other data. *Lambda abstractions*, from the formal λ-calculus [4], define nameless expressions of function type, and they constitute "function typed values". They have the form $\lambda x.e$, where x, a variable, is "formal parameter" and e, an expression, is the "function body". If x has type α and e type β then $\lambda x.e$ has type $\alpha \to \beta$, and it can be applied to any expression e' of type α. For instance, $\lambda x.(x+1)$ is a nameless version of the function $f(x) = x + 1$.

An expression e of type $\alpha \to \beta$ can be given a meaning $[\![e]\!]$ as a mathematical function $[\![\alpha]\!] \to [\![\beta]\!]$, where $[\![\alpha]\!]$ and $[\![\beta]\!]$ are interpretations of the types α, β as *complete partial orders* (c.p.o.'s, see, for instance, [42]). A c.p.o. contains a *least element*, which is not a computable value but rather represents divergence or "complete lack of information". If f is a recursively defined function, then $f(x) = \bot$ usually means that the computation of $f(x)$ does not terminate (which is truly lack of information, since no result is ever delivered).

An n-ary function f is *strict in argument i* if $f(\ldots, x_{i-1}, \bot, x_{i+1}, \ldots) = \bot$ for alla possible values of x_j, $j \neq i$ (i.e., given a nonterminating i:th argument, its evaluation will not terminate). Otherwise it is *nonstrict in argument i*. Arithmetical operations are usually strict in all their arguments. The *if*-function, defined by

$$\begin{aligned} \mathit{if}(\mathit{true}, x, y) &= x \\ \mathit{if}(\mathit{false}, x, y) &= y \\ \mathit{if}(\bot, x, y) &= \bot \end{aligned}$$

is strict in its first argument but not in its second (since $\mathit{if}(\mathit{false}, \bot, y) = y$) and third.

Functional languages differ in how they treat arguments to user-defined functions. Most languages use *eager evaluation* (or *call by value*), meaning that the arguments are fully evaluated before the function call takes place. In such languages, user-defined functions are always strict in all their arguments. Some languages use a more complicated strategy known as *lazy evaluation* (or *call by need*), where the evaluation of an argument is deferred until its value is actually needed. In lazy languages, user-defined functions can be nonstrict in some arguments. Lazy languages offer the possibility to use infinite data structures, e.g.,

streams. They also have a theoretical advantage in that they have a very simple abstract semantics. But they require elaborate parameter passing mechanisms, which can be costly.

Every recursive function definition $f(x) = E$ can be seen as an equation. Indeed, one can say that *functional programming is programming with equations*. Functional languages have found their use mostly in highly symbolic computing, like in compilers, interpreters, or theorem provers. There are several reasons for this. For one, this kind of computation typically involves complex data structures, and many functional languages offer excellent support for this in the form of *abstract data types*, i.e., data types specified by axioms. For instance, lists can be specified as terms built out of the "constructors" *nil* (empty list) and "::" ("cons"), and the decomposing functions *head* and *tail* can be specified by the equations

$$head(x::L) = x$$
$$tail(x::L) = L$$

Equational axioms like this fit very well into the functional paradigm and can often be directed into computation rules right away.

However, computations involving complex data structures are not the only ones naturally specified by equations. Another area is numerical computing, where the algorithms often are specified by recursion equations which can be interpreted more or less directly as functional programs. But functional programming has not catched on for this class of computations. Again, there are several reasons: performance problems (despite simpler parallelization) and unfamiliarity among programmers are two of them. Another, more subtle reason is that convenient programming for numerical computing often requires a different notion of aggregate data than the data structures for symbolic computing, and these notions are not well supported in all functional languages. Many functional languages, for instance, lack support for anything except basic array operations. (A notable exception is Sisal [24].)

4 Data Parallel Functional Languages

Despite the dominance of functional languages tuned for symbolic computing applications, there have been attempts to incorporate array and data parallel operations in functional languages. In particular, there are a number of languages emanating from the efforts in data flow computing in the eighties. The traditional way to add data parallelism is to designate a certain aggregate data type for this purpose, and to provide (1) a way to define entities of this type, and (2) a number of primitives operating on these.

Arrays are a common carrier of data parallelism. Sisal [24, 50] is a strongly typed, call-by-value functional language intended for scientific computing. It has one-dimensional arrays (multi-dimensional arrays are thus expressed as arrays of arrays). Arrays can be defined through a `For` construct which specifies a range

and, for all indices in that range, the value of the corresponding array elements. Thus, it is somewhat like a side-effect free version of the `FORALL` construct of HPF. There is a rich set of array operations. The design of Sisal is made to enhance the generation of efficient code, and Sisal has efficient implementations on a number of vector- and parallel architectures [12].

Equational Programming Language [43, 53] has recursive array definitions with strong syntactic restrictions ensuring that they can be interpreted as *recurrent equations*. This enables the use of powerful program analysis methods and optimizing compilation. The Id Language [22] has *array comprehensions*, which serve the same purpose as the `For` construct of Sisal. They are, however, more general in that they are *non-strict*, which means that only the part of an array that is actually used is computed. This is essentially lazy evaluation applied to arrays, and allows array definitions with nonterminating elements. Id array comprehensions can furthermore be *recursive*, i.e., an array can be defined in terms of itself.

Array comprehensions are also found in the lazy language Haskell [35]. The array elements are defined by a list of *associations*, i.e., pairs of indices and element values. This list is often given as a *list comprehension* (i.e., a convenient syntax for lists which allows *enumerations* over integer intervals, akin to loop ranges). It is possible to leave out certain indices within the bounds: the corresponding array elements are then undefined. Haskell array comprehensions are not primarily intended for parallel implementation, but there is nothing that prevents it [1]. Data Parallel Haskell (or DPHaskell) [31] is a version of Haskell extended with "Parallel Objects of Arbitrary Dimensions" (PODs). These are essentially nonstrict arrays, possibly without bounds, supporting parallel operations. Unbounded, infinite PODs are possible thanks to the nonstrictness. Another feature is the use of *guards*: boolean expressions that filter out certain elements of PODs. This makes it possible to express sparse PODs.

NESL [7, 8], a successor to Paralation Lisp [9] with a similar parallel data type, is a strongly typed, first order functional language that uses nested *sequences* as parallel data. A sequence in the sense of NESL is essentially a one-dimensional array indexed from zero and up. Thus, the nested sequences of NESL are quite close to the nested arrays of Sisal. NESL is implemented on several parallel architectures on top of the intermediate format VCODE [15].

The language Crystal [17] adds another level of abstraction: parallel data are considered as *functions* ranging over finite *index domains*. These domains are constructed from a number of base domains (integer intervals, hypercube coordinates, trees) which can be combined using constructors for product, direct sum and function space. The definitions can be recursive. In [59], a *restriction* operation on index domains is described that is similar to DPHaskell's guards. A similar but more restricted language is Alpha, for which efficient compilation techniques have been developed [46, 47]. These methods are based on *space-time mappings* [16, 20, 33, 37, 39, 40, 44, 45, 48, 49] of certain recurrent equations, which statically specify where and when each step in the recurrence will be computed.

A similar view of parallel data appears in Connection Machine Lisp [52]. Here, the parallel data type is the *xapping*, which is a set of pairs of Lisp objects where the first component of a pair cannot occur in another pair. Thus, xappings are really set-theoretical functions.

4.1 Formalisms

There are a number of functional formalisms which can be used as an abstract programming notation, to specify data parallel algorithms on a mathematical level, or to identify and prove algebraic laws which can be used for program transformations. Some of these formalisms are executable and have been implemented, while others can be used only as an abstract notation.

An early example is Backus' FP [2]. FP is a formalism entirely based on functions and operations on functions, most prominently function composition. These operations are called *functional* (or *combining*) *forms*. FP supports a "combinatory style" of specification, i.e., functions are defined just by combining other functions, without using formal parameters. Indeed, FP does not have lambda abstraction. Values in FP are either *atoms* or *tuples* (sequences): certain functions operate on sequences and can then often be seen as data parallel. For instance, FP has a *Map* operation that serves as elementwise application, and *Right Insert* and *Left Insert* which can be used for reduction over sequences. FL [3] is an implementation of FP.

Similar to FP in spirit is the *Bird-Meertens formalism*. Here an algebra with unary and binary functions forms a base for a set of theories for different data types [5, 6]. In particular, there is a theory for functions over lists. The lists can be seen as carriers of parallel data; then, the formalism gives a framework to specify and reason about data parallel algorithms. The Bird-Meertens formalism has *Map* and *Reduce* operations which can be directly interpreted as data parallel operations, as well as a *Filter* operation which forms a "compressed" list of the elements for which some predicate is true. Skillicorn [51] has shown that the Bird-Meertens formalism provides a model of data parallelism that is *universal* over a broad class of parallel architectures, i.e., that there is a cost model which gives the correct order of magnitude for the cost of executing the primitives in the formalism on these architectures.

A different approach is Carpentieri's and Mou's algebraic model for *Divide-and-Conquer algorithms* [14]. Here, Divide-and-Conquer is expressed as a certain form of recursion. The parallel functional programming language *Divacon* [13] is based on this model: it is targeted towards efficient implementation of Divide-and-Conquer algorithms on hypercube SIMD architectures.

The PEI (Parallel Equations Interpreter) formalism [57, 58] is intended to specify recurrent equations. The central concept is the *data field* (not to be confused with data fields as defined here), which is a pair of a partial function v from tuples of integers and a bijection σ from the same tuples. v can be seen as a data field in our sense, and σ can be interpreted as a space-time mapping. PEI also includes a *refinement calculus*, which makes it possible to perform formal derivations of parallel algorithms for the recurrences from looser specifications.

Finally, APL [36] must be mentioned since it probably is the first example of a formalism (and programming language) where the typical data parallel constructs appear.

5 An Abstract Formalism for Aggregate Data

All the languages and formalisms in the previous section have in common that they use a single carrier of parallel data. But data parallel operations are clearly definable and useful over a wide range of data fields (e.g., array, list, sparse, etc.). If the natural index domain of a problem does not match the carrier of the language well, then the programmer must coerce the index domain into the format of the carrier. Thus, if expressiveness is a primary concern, then it is desirable to have languages and formalisms which capture the essence of data parallelism on a still higher level of abstraction. In the following we attempt to define such a formalism, to explore its potential to serve as a base for data parallel programming languages, to demonstrate its advantages, and to make viable that nontrivial languages based on this formalism can be implemented in a reasonable way.

5.1 Data Fields

The basis for our formalism is the simple observation that indexed structures are really *partial functions*, e.g., an array with range 1..10 is a partial function from integers defined only for arguments $\in \{1, \ldots, 10\}$. We can then model partial functions as total functions over c.p.o.'s, which return $\perp$ when the partial function is undefined.

Definition 1. A *data field* is a function $D \to D'$, where D and D' are c.p.o.'s. The *index set* of the data field f is $dom(f) = \{ x \mid f(x) \neq \perp \}$.

This view of aggregate data is not tied to any particular data type, since we have not specified D and D'. To model arrays, D can be choosen as the c.p.o. $[\![int]\!]$ of integers. Also for lists, we can pick $D = [\![int]\!]$, since a list has an implicit indexing of its elements. For n-dimensional arrays, $D = [\![int]\!]^n$. Nested arrays, e.g., arrays of arrays, can be considered as "curried" functions $[\![int]\!] \to ([\![int]\!] \to D')$. Data structures can be given an abstract indexing very close to the problem domain, but the indexing can also model the data distribution for an implementation by choosing D close to the memory model of the target architecture (e.g., pairs of processor identifiers and local memory addresses).

Our formalization of data fields yields the "individual index set" programming model, where every data parallel entity carries its own index set. If all elements of a data field are requested, then it suffices to compute the elements defined over the index set. It is therefore of great interest to have methods to decide whether the index set is *finite*, since that allows computation of the data field in finite time. We do not exclude infinite data fields, though: they can make very good sense in *lazy* languages, where only the requested part will be evaluated.

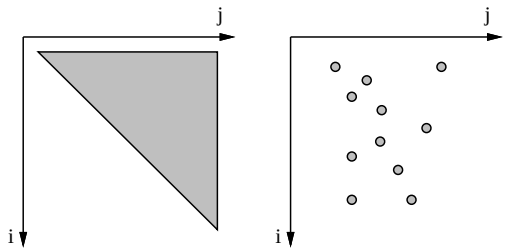

Fig. 6. Two possible data fields over $[\![int]\!] \times [\![int]\!]$: a triangular field and a sparse field.

5.2 A Formal Language Scheme

Our formalism concerns expressions in a formal language. These expressions define partial functions which in turn can be interpreted as data structures. Formal languages for partial functions can be defined in many ways: our choice aims at minimizing the number of language constructs while being as close as possible to familiar concepts. We do not specify all the details completely; rather, we present a *language scheme*, with the following properties:

- Typed, with basic types and type constructors "$\times$" (pair) and "$\rightarrow$" (function).
- A number of strict operations over the basic types.
- For every basic type a constant "$\perp$" representing the bottom element.
- For every type a conditional, with interpretation as the *if*-function defined in Section 3.
- λ-abstraction, i.e., if t has type τ and x has type σ, then $\lambda x.t$ has type $\sigma \rightarrow \tau$.

We use the n-ary notation $\lambda x_1 \cdots x_n.t$, where x_i has type σ_i, to denote a function of type $\sigma_1 \times \cdots \times \sigma_n \rightarrow \tau$ (rather than the curried function of type $\sigma_1 \rightarrow (\cdots \rightarrow (\sigma_n \rightarrow \tau)))$.

The basic types will typically be *int*, *float*, *bool*, etc. Other constructed types, such as lists and disjoint union types, can also be added but are not necessary. This is true also for recursive definitions.

In short, we have a typed language with ordinary basic types and operations, a conditional, higher order expressions through λ-abstraction, and a distinguished constant representing domain-theoretic bottom. Everything except the last one is readily expressible in a conventional, higher order functional language. In Section 6 we will look into issues for real languages based on this formalism; let us already now point out some features such languages will have:

- The data parallelism will not be a priori confined to a single data type.
- There is no need to introduce new operations on parallel data, λ-abstraction plus the first order operations and constants will suffice.
- Considering aggregate data as partial functions adds a new level of abstraction, which gives an implementation freedom to choose the most appropriate underlying representation for a particular machine.

A data field definition in a language according to the above will be a function definition. The only new language element is $\perp$. Thus, it is straightforward to give data field definitions a denotational semantics (i.e., an interpretation as a mathematical function). It can for instance be the least fixpoint solution to the definition.

5.3 Explicit Restriction

It is convenient to introduce a derived restriction operation: we expect it to play a major rôle when it comes to real programming, and furthermore a number of *algebraic laws* can be proved which involve this operation. See Section 5.5.

Definition 2. For any data field $f\colon D \to D'$ and boolean data field $b\colon D \to [\![bool]\!]$, $f \setminus b = \lambda x.\mathit{if}(b(x), f(x), \perp)$.

So, for instance, if f is a function $[\![int]\!] \to D'$, then $f \setminus \lambda i.(0 \leq i < n)$ is a function $[\![int]\!] \to D'$ that is guaranteed to be undefined for any argument i that falls outside the range $0 \leq i < n$, and so, if seen as a data field, it makes sense to compute its values only for arguments within that range. This particular use of explicit restriction mimics array comprehensions. In general, restriction w.r.t. linear inequalities yields array-like data fields; see, for instance, Fig. 6, which pictures a triangular data field restricted by the predicate $\lambda ij.(1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \leq j)$. Note, however, that we allow *any* predicate b in Definition 2. So we could for instance define a "sparse array data field" $f \setminus (\lambda i.(0 \leq i < n) \wedge b)$, where b is any predicate. Thus, explicit restriction can be given a dual rôle: both to give "bounds" for array-like data fields, and as the restriction found in other data parallel languages.

Obviously, it is interesting to decide whether a predicate occurring in a restriction is true in a finite number of points or not, since finiteness implies that the index set of the restricted data field is finite. Unfortunately, this is undecidable for general predicates due to the halting problem. But for certain classes of predicates it is decidable. An important case is systems of linear inequalities, where the finiteness is equivalent to emptiness of an associated system of linear inequalities. This can be decided by polyhedral methods: for an introduction, see [23].

It is sometimes convenient to define a "scalar" version of the restriction operator, viz. $x \setminus y = \mathit{if}(y, x, \perp)$. We then have $f \setminus b = \lambda x.(f(x) \setminus b(x))$, i.e., the scalar restriction is *elementwise applied* to f and b according to Definition 3 below.

5.4 Data Parallel Operations

Operations on data fields are operations on functions. The interpretation is that an operation on a data field can be thought of as a (conceptually parallel) computation, where all the defined values of the resulting function are requested. We have already considered explicit restriction. We will now show how the other data parallel operations of Section 2 can be expressed as operations on functions.

Elementwise Applied Operations

Definition 3. If g has type $\gamma_1 \times \cdots \times \gamma_n \to \tau$ and if f_i, $1 \leq i \leq n$ have type $\sigma \to \gamma_i$, then $\lambda x.g(f_1(x), \ldots, f_n(x))$ is the *elementwise application* of g to $f_1, \ldots, f_n$.

Here, g is the elementwise applied operator and the f_i are interpreted as data fields. The interpretation is that $g(f_1(x), \ldots, f_n(x))$ is computed for any x where it is defined. Thus, elementwise application is merely a form of function composition (see Fig. 7).

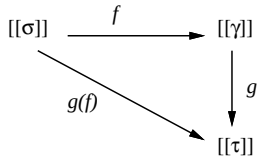

Fig. 7. $g: \gamma \to \tau$ elementwise applied to a data field $f: \sigma \to \gamma$.

λ-abstraction can be thought of as a side-effect-free *Forall* statement. Indeed, one could use the syntax *Forall* x $g(f_1(x), \ldots, f_n(x))$ for $\lambda x.g(f_1(x), \ldots, f_n(x))$.

It is possible to introduce an overloaded syntax, corresponding to the Fortran 90 array operations, provided that enough type information is available. If we can uniquely infer that g has type $\gamma_1 \times \cdots \times \gamma_n \to \tau$ and each f_i has type $\sigma \to \gamma_i$, then we can write $g(f_1, \ldots, f_n)$ and interpret this unambiguously as $\lambda x.g(f_1(x), \ldots, f_n(x))$. For instance, if both f_1 and f_2 have type $\tau \to int$, then $f_1 + f_2$ is "syntactic sugar" for $\lambda x.(f_1 + f_2)$. Note that this overloading relies only on type information and is not restricted to intrinsic operations as in Fortran 90, it is just a way to denote function composition. It is also possible to extend this general overloading to promotion of scalars, see Section 5.5.

Elementwise application of the *if*-function can be used to "mask out" parts of data fields and "splice" the result, e.g., $if(f > 0, f, -f)$ is a side-effect-free way expression for the elementwise absolute value of f. Cf. Section 2.4.

Another example is *array concatenation*. Let (one-dimensional) *arrays* of type τ be defined as functions $f: int \to \tau$ with an attribute $length(f): int$ such that $dom(f) \subseteq \{\, i \mid 0 \leq i < length(f) \,\}$. (That is, an array of length n is indexed from 0 to $n-1$.) Array concatenation "$\cdot$" can now be defined by

$$f \cdot g = \lambda i.if(i < length(f), f(i), g(i - length(f)))$$
$$length(f \cdot g) = length(f) + length(g)$$

Array concatenation involves a shift of the second array. This is an instance of get communication, see below.

Communication Operations Get communication is simple to model in our formalism, since it does not involve side effects:

Definition 4. If f is a function $\tau \to \sigma$ and g is a function $\gamma \to \tau$, then $\lambda x.f(g(x))$ is the *get of f from g*.

Thus, get communication is also function composition, but to the "right", see Fig. 8. The intuition is that for each x, the value of f at "source address" $g(x)$ is fetched. If the elements of f and $\lambda x.f(g(x))$ are stored in a distributed fashion, then g specifies a concurrent read from the processor of $g(x)$ to the processor of x, for all x where $f(g(x))$ is defined. As for elementwise application, the overloaded syntax $f(g)$ can be used to denote $\lambda x.f(g(x))$.

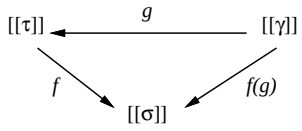

Fig. 8. Get communication with "source data" $f: \tau \to \sigma$ and "source address map" $g: \gamma \to \tau$.

Send communication, as specified in Section 2.2, has side effect and is thus not as straightforward to model in our framework. It is, however, possible to define a side-effect-free version of combining send as the *segmented reduction* in [28, 29].

Replication Replication is easily specified through λ-abstraction. If y has type σ, then we can create a "constant valued" data field $\lambda x.y$ of any function type $\tau \to \sigma$ by abstracting with respect to a fresh variable x of type τ. This data field is defined everywhere and thus infinite: an explicit restriction can give it a finite range. In a lazy setting the actual range can also be given implicitly by the context. Correspondingly, a "one-dimensional" data field f can be replicated into "matrices" $\lambda ij.f(i)$ ("replicate as columns") and $\lambda ij.f(j)$ ("replicate as rows").

Promotion of scalars to data fields can be done through type inference, in a way akin to the overloading of scalar operations. Any time a scalar y of type σ occurs in a context where an expression of type $\tau \to \sigma$ is expected, it can be automatically promoted into $\lambda x.y$ where x is a fresh variable of type τ. For instance, if f has type $\tau \to int$, then the expression $f + 17$ can be translated into $f + \lambda x.17$, which can be further translated into $\lambda x.(f(x) + 17)$.

Reductions Reductions and related operations can be defined as higher order operations through recursion. (They can be seen as *algorithmic skeletons* [19].) Below, we define a general reduction which can reduce any data field $f: \tau \to \sigma$ with respect to any binary operation $g: \sigma \times \sigma \to \sigma$. Since we don't make any

assumptions about the algebraic properties of the binary operation our reduction needs also an *enumeration*, $i\colon int \to \tau$, that specifies the reduction order of the elements of f.

Definition 5. $red\colon ((\tau \to \sigma) \times (\sigma \times \sigma \to \sigma) \times (int \to \tau) \times int) \to \sigma$ is defined by:

$$red(f, g, i, n) = if(n = 1, f(i(0)), g(red(f, g, i, n-1), f(i(n-1)))).$$

In other terms, with an infix binary operation "$\circ$",

$$red(f, \circ, i, n) = (\cdots(f(i(0)) \circ f(i(1))) \cdots) \circ f(i(n-1)).$$

n is the number of elements to reduce over. This reduction of Definition 5 is not parallel. But if g is associative, then it is easy to show that this definition is equivalent to a balanced recursion that yields $O(\log n)$ recursion depth.

If the operation defined above is to implement what is usually perceived as reduction, i.e., an operation using all the defined elements in f but no others, then the arguments must have certain properties. First, n should equal the number of elements in the index set of f. Second, the function i should really be an enumeration of the elements in the index set, that is: a bijection from $\{0, \ldots, n-1\}$ to $dom(f)$. To assert these properties is a matter of program verification, they are not guaranteed by the definition alone.

Definition 5 is very explicit since it requires the enumeration and the number of elements as arguments. This is often not convenient. In many circumstances, one can define less explicit forms. For instance, one can define a function $red\_int(f, g, n) = red(f, g, \lambda x.x, n)$, for data fields $f\colon int \to \sigma$, that reduces f in a left-to-right fashion on the interval $0, \ldots, n-1$. The ordering can also be given implicitly by the context, or be irrelevant (say, if g is associative and commutative). But then there must be some other way to find the n elements to reduce over, since they are not explicitly given in the recursion anymore. Finding these will in general require operations which are able to distinguish between "divergent $\perp$" and "out of range-$\perp$", see Section 6.5.

Other reduction operations, like scan and the segmented variants of reduction and scan, can be defined similarly to Definition 5. See [28, 29].

5.5 Algebraic Laws

We will now present a number of algebraic laws for the explicit restriction. They can be seen as constraint propagation rules, and they are clearly of use for, e.g., program optimization. For instance, they can guide the preallocation of memory, and help reduce the amount of computation, by predicting statically which parts of data fields will be defined and which will not.

All the results below follow more or less immediately from Definitions 1 and 2. "$\wedge$" and "$\vee$" refer to the non-strict versions of conjunction and disjunction, respectively, for which $false \wedge \perp = false$ and $true \vee \perp = true$. All operators are elementwise applied below: we use overloaded syntax.

Proposition 6. *$(f \setminus b) \setminus b' = f \setminus (b' \wedge b)$.*

Proposition 7. *$(f \setminus b)(g) = f(g) \setminus b(g)$, and if g has a left inverse g^{-1}, then $f(g) \setminus b = (f \setminus b(g^{-1}))(g)$.*

For elementwise application there are a number of identities. The first essentially says that an outer restriction always can be "pushed" to the arguments of an elementwise applied scalar operation:

Proposition 8. *$g(f_1, \ldots, f_i, \ldots, f_n) \setminus b = g(f_1, \ldots, f_i \setminus b, \ldots, f_n) \setminus b$ for any i from 1 to n.*

The following identity allows the propagation of conditions from strict arguments "outwards":

Proposition 9. *If g is strict in argument i, then $g(f_1, \ldots, f_i \setminus b, \ldots, f_n) = g(f_1, \ldots, f_i, \ldots, f_n) \setminus b$.*

Corollary 10. *If g is strict in all its arguments, then $g(f_1 \setminus b_1, \ldots, f_n \setminus b_n) = g(f_1, \ldots, f_n) \setminus (b_1 \wedge \ldots \wedge b_n)$.*

See Fig. 9: in other words, a data field given by an elementwise applied strict operation is defined on the intersection of the index sets of the arguments. This has interesting consequences for array languages. For instance, array arguments to elementwise applied operations need not necessarily be conformant, the resulting array will be defined on the intersection of the argument ranges anyway. One can go even further and allow, e.g., sparse and "dense" data fields in the same expression: Corollary 10 guarantees that the resulting data field is well-defined no matter the shape of the argument fields.

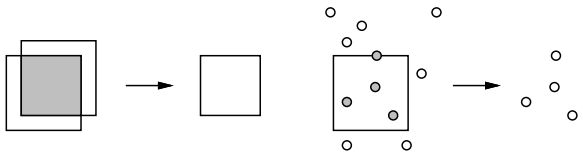

Fig. 9. Applying a strict binary operation elementwise to: two matrix-like data fields, and a matrix-like and a sparse data field.

Proposition 8 and Corollary 10 also support promotion of scalars to data fields with automatic conformation to fit the given context. Consider, for instance, the expression $f \setminus b + 17$. Promotion of scalar turns this into $f \setminus b + \lambda x.17$, and by the identities this equals $(f \setminus b + \lambda x.17 \setminus b) \setminus b$. Thus, even though $\lambda x.17$ is unrestricted, its use will be restricted to the same points as where $f \setminus b$ is defined, i.e., it fits exactly the extent of $f \setminus b$.

Finally, for the elementwise applied conditional function, we have the following identities (see also Fig. 10):

Proposition 11.

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

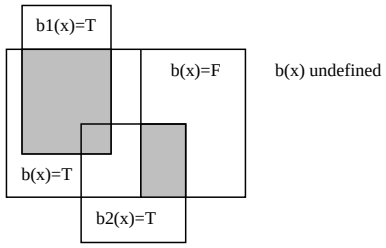

Fig. 10. Elementwise application of the *if*-function. $\mathit{if}(b, f \setminus b_1, g \setminus b_2)$ is defined on the shaded areas.

6 Towards Implementations

In this section we will argue that purely functional data parallel languages can be almost directly defined according to the formalism developed in Section 5. Such languages can be based on higher-order strongly typed languages: using functions as data fields in the way suggested by our formalism will then require a very small extension of the language. A more detailed account of the material presented here is found in [29].

6.1 C.p.o.'s with Explicitly Undefined Values

In our formalism the abstract value $\perp$ is really used for two different purposes: to denote nontermination, and to represent a failing lookup. The second situation can be detected in finite time and is thus in some respect a "soft" failure. Operationally, these situations are very different and it can make good sense to distinguish between them: for instance, we would not like program transformations to turn a "terminating $\perp$" into nontermination. However, without some means to distinguish between these behaviours we cannot guarantee that this cannot happen!

Thus, a more detailed semantics is needed that takes this difference into account. This can be accomplished by introducing, for every semantical c.p.o., a novel element "$*$" which means "explicitly undefined" and represents a failing lookup. $*$ should behave algebraically as $\perp$, i.e., if $f(\ldots, \perp, \ldots) = \perp$ then we

should (at least, preferrably) have $f(\ldots,*,\ldots) = *$. We can think of $*$ as a kind of error value (and, conversely, error values for other purposes, such as divide-by-zero, should have similar properties).

Semantically, we replace the old interpretations $[\![\tau]\!]$ of types τ, as c.p.o.'s which do not distinguish explicit undefinition and divergence, with new interpretations $[\![\tau]\!]_*$ as c.p.o.'s with explicitly undefined elements $*$. This means that also the operations in the language in question must be given new interpretations as functions over the new c.p.o.'s, i.e., they must be extended to deal with the case when some argument(s) is $*$.

How should functions over old interpretations $[\![\tau]\!]$ be extended to interpretations $[\![\tau]\!]_*$ with explicitly undefined elements? A function $f_*: [\![\tau]\!]_* \to [\![\tau]\!]_*$ is a *consistent extension* of $f: [\![\tau]\!] \to [\![\tau]\!]$ if it, roughly speaking, obeys the same algebraic identities with respect to $*$ as the ones f obeys with respect to $\bot$ (for an exact definition, see [29]). Being a consistent extension is not a fundamental property, and indeed there are useful functions over $[\![\tau]\!]_*$ which are not (see Section 6.5), but the consistent extensions have certain nice properties. For instance, most[1]of the algebraic laws in Section 5.5 generalize to consistently extended functions, which means that program transformations based on these laws will preserve also the termination properties!

In the following we will use the if_*-function, which is the conditional consistently extended to handle $*$ according to the following axiom:

$$if_*(*, x, y) \quad = \quad *$$

We can use this conditional to define an extended explicit restriction:

$$f \setminus_* b = \lambda x.if_*(b(x), f(x), *).$$

From now on, we will assume that all operators of the assumed language (with the possible exceptions of Section 6.5) have a consistently extended interpretetation.

6.2 Parallel Evaluators

So far, we have been talking loosely about data fields as an interpretation of function-typed expressions as partial functions. But what do we mean? Actually, we are talking about a *different evaluation mechanism for functions*. The conventional evaluation mechanism for functions which is used in traditional functional languages is based on the β-reduction of the λ-calculus [4], which means that a function is not evaluated until it is applied to an argument. So a function can only be computed "pointwise", for one argument at a time. The

[1] The exception is Corollary 10, which holds in a weaker form for "reasonably" consistently extended strict operations. The problems arise for extensions of operations which are strict in more than one element. These can be consistently extended in various ways, differing in how they treat the cases where some strict arguments are $*$ and others are $\bot$. Some of these extensions will require expensive implementations, those which do not will obey weaker forms of Corollary 10. See [29].

"data field reading" of a function is different: it implies an attempt to evaluate *all* the possibly defined values of the (partial) function in order to *tabulate* it. This is a (conceptually) *parallel* evaluation mechanism, and it is different from the conventional one.

Explicit control over which mechanism to use is given by introducing a particular operator that forces a "parallel evaluation" of a function-typed expression. We call this operator "*request*", since it requests all the possibly defined values of a data field. Thus, *request*(f) will, if it succeeds, yield a data structure which holds a table over the defined values of f. So *request*(f) is also a function, albeit tabulated, and subsequent calls to *request*(f) will be implemented by some kind of lookup.

An ideal *request* operator would always find the exact index set of its argument and then tabulate the argument for those points. But such an ideal operator cannot exist for other than trivial functional languages. This limitation is due to the halting problem: the *request* operator itself may fail to terminate when applied, and there is in general no mechanism to predict when this will happen. This means that there is no "best" way to define *request*: any solution will have to be approximate, and there will be a tradeoff between the accuracy and the complexity of the index set approximation procedure. Indeed, one can very well have *several* different *request* operators in the same system which provide different tradeoffs. However, they should all behave according to the following scheme:

1. Attempt to find a set S of total values (not containing $\perp$), such that $x \notin S \implies f(x) = *$.
2. Enumerate S.
3. Evaluate f for all $x \in S$.
4. If terminating, return a data structure with a table of the resulting values.

Condition 1 guarantees that the lookup procedure, which is used to apply *request*(f) as a function, safely can return $*$ when the lookup fails. Ideally, S should be finite whenever $dom(f)$ is, but as reasoned above this cannot be achieved in general. A variant of the scheme above tests whether S is finite in order to avoid nontermination due to an infinite search for values to tabulate. However, as reasoned above, mere finiteness of S is not sufficient for tabulation: we must also know an *enumeration* of the elements in order to create the table of values.

Also note that *request*(f) will fail to terminate if $f(x) = \perp$ for any total x, that is: *request* is *hyperstrict* [56]. The situation is akin to the evaluation of a list-valued expression at the top level in a lazy functional language. Indeed, the eval-print loop of such a language can be thought of as using an equivalent to *request* in order to force the evaluation of all the elements in the list.

How could a *request* operator go about finding an S? There are several possibilities. A crude *request* could require that its argument be given on the form $f \setminus b$. It could then inspect the predicate b to see whether it defines a suitable S or not. For instance, it could deem as acceptable predicates of the form

$\lambda i_1 \cdots i_n.(l_1 \leq i_1 \leq u_1 \wedge \cdots \wedge l_n \leq i_n \leq u_n)$, where the l_j and u_j are integers: this kind of predicate specifies array-like data fields and a *request* operation accepting such predicates will act very much like the array comprehensions of, e.g., Haskell [35]. See Fig. 11. Other formats are certainly also possible to recognize. An immediate generalization is to allow b to be a general system of linear inequalities: the problem to enumerate the integral points in polyhedra defined by linear inequalities is exactly the *loop scanning problem* [23, 38]. As mentioned in Section 5.3, there are also methods to decide the finiteness of polyhedra.

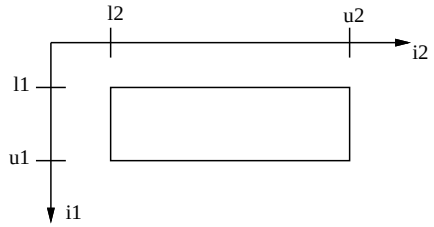

Fig. 11. An array-like restriction.

It is also possible to envisage *request* operators which perform a more elaborate search for a "narrower" S. Such an operator can for instance utilize the algebraic laws of Section 5.5. In particular, the laws for elementwise application of strict operators are useful since they lead to the taking of intersections. Certain classes of predicates, e.g., the "array predicates" exemplified above and also the more general systems of linear inequalities, define classes of sets which are closed under intersection. A prototype implementation of a simple functional language with a *request* operator performing a search of this kind exists [27].

6.3 Loosely Specified Reductions

For binary operations which are both associative and commutative, the order of operations and operands is not significant. For such binary operations (or, when an ordering of the operands is implicitly given), it is convenient to have a loosely specified reduction $red(f, g)$ which takes as argument only the data field f to reduce over and the binary operation g. Such a reduction can be based on *request* as follows:

1. Compute $request(f)$ (over the set S).
2. Reduce over the resulting values using the enumeration of S computed by *request* (or some implicitly given ordering).
3. Return the result.

The set S computed by *request* can be approximate, i.e., we may have $f(x) = *$ for some $x \in S$. These elements must be sorted out before reducing (or else, if

g is strict, the reduction will yield $*$). This means, at some level, that we need a way to test for $*$ that returns *true* or *false* rather than $*$. Such a test must be a non-consistent extension, se further Section 6.5.

6.4 Extent Analysis

It is sometimes possible to analyze data field expressions at compile time in order to find properties which allow the generation of simpler but still correct code. This is particularly valuable in order to avoid the runtime overheads associated with advanced evaluation mechanisms, like elaborate *request* operators. *Extent analysis* [41] refers to two static analyses, formulated as abstract interpretations [18], which are applicable to data field expressions. "Extent" stands for index sets or needed parts thereof: in these analyses the extent of a function $[\![\sigma]\!] \to [\![\tau]\!]$ is represented by a predicate $[\![\sigma]\!] \to [\![bool]\!]$.

Input-output analysis tries to infer, given the extents of the "input" data fields, an approximation $f^\#$ to the index set of the resulting data field f. $f^\#$ has the property that $f(x) \neq \bot \implies f^\#(x)$: thus, it provides a safe approximation to the index set of f. (This analysis, as given in [41], is formulated with respect to the simpler semantical c.p.o.'s where $*$ is identified with $\bot$: it is straightforward to reformulate it to c.p.o.'s with $*$ and prove $f(x) \neq * \implies f^\#(x)$.)

Input-output analysis can be used to perform the analysis phase of *request* at compile time, since $f^\#$ yields a possible S. This can enable static allocation and scheduling methods which can be used to produce efficient code.

Output-input analysis tries to find, for a predicate b, data field expression t and data field f used to evaluate t, the points x where calls will be made to $f(x)$ in order to evalutate $t(x')$ in any point where $b(x')$ is true. Thus, the result of a successful analysis can be used to preschedule calls to f given that *request*$(f \setminus b)$ is to be evaluated.

Input-output analysis propagates constraints from input data fields to results, whereas output-input analysis propagates constraints on outputs (requested parts of data fields) to inputs. Thus, they are related to the Floyd-Hoare semantics of imperative languages, in particular the *strongest postcondition* and *weakest precondition* predicate transformers [21]. Directly related are methods to analyze constraints on integer typed variables, e.g., for the purpose of analyzing array references statically. In particular, the polyhedral abstract interpretation of Halbwachs [26] seems relevant.

6.5 Non-Consistent Extensions

There is at least one non-consistent extension that seems useful. Sometimes it makes sense to have a special equality predicate "$='$" where $* =' *$ returns *true* and otherwise *false* is returned. Since $=$ is strict, any *consistent* extension must return $*$ as soon as any operand equals $*$: thus, $='$ is a non-consistent extension. Nevertheless, it is perfectly computable and can be provided as a primitive.

For instance, $='$ can be used to define a "*fill*" operation that fills in something where a data field is explicitly undefined:

$$\mathit{fill}(f, g) = \lambda x.\mathit{if}_*(f(x) =' *, g(x), f(x)) \tag{1}$$

This operation can be used to add a sparse data field g into a dense field f in exactly the points where g is defined, viz:

$$f + \mathit{fill}(g, 0).$$

The result is a data field with the same extent as f. Note that $f + g$ would be sparse.

6.6 Implementations of $*$

Our programming model, with an explicitly undefined element $*$, requires that we represent that element somehow and extend the operations in the language to deal with it. This can be done in several ways, here are a few:

The naïve way is to give $*$ a tagged representation and extend operations to handle $*$ through explicit tests for $*$. For instance, extended addition (denoted "$+_*$") can be implemented as

$$+_* = \lambda xy.\mathit{if}_*(x =' * \vee y =' *, *, x + y).$$

This kind of implementation is, however, extremely inefficient on most processors since it contains a conditional at a very fine-grain level.

A better alternative, which is sometimes possible, is to make an exception-based implementation where occurrences of $*$ are catched, the execution is interrupted and $*$ is returned. This is correct for consistently extended operations since these should return $*$ whenever a strict argument is $*$. With this kind of implementation, $*$ will behave very much like "failure" in Prolog. For details, see [29].

For floating point computations according to the IEEE standard the "NaN" (Not a Number) entities can represent $*$, since they have the correct algebraic properties [25].

When operating on data fields, it can be advantageous, for a data field f, to have a "aggregated tagged" representation as a bit mask b paralleling the data structure with the "real" values of f, such that $b(x) = \mathit{false}$ encodes $f(x) = *$. Elementwise applied strict operations can be efficiently implemented in this way, by first ANDing the bit masks of the operands elementwise and then applying the operation only in the points where the result is *true*. SIMD machines provide hardware support for this through the local context flags, which control the participation of processing elements.

7 Conclusions and Further Research

We have presented a formalism for data parallelism where aggregate data are considered to be partial functions, and operations on aggeregate data are higher order operations on functions. Formal languages for this can use the classical λ-syntax extended with constants, and in particular a constant "$\perp$" representing domain-theoretic bottom is needed.

We demonstrated that a number of common data parallel operations could be expressed in such a language, and we proved algebraic laws which can be used to transform expressions specifying aggregate data in order to simplify the computation of these. We then proceeded to show that data parallel programming languages can be based directly on the formalism. For this, we needed to introduce more refined semantical c.p.o.'s, distinguishing between the divergent $\perp$ and explicit undefinition, and a *request* operation creating an environment where function-typed expressions are given an interpretation approximating the set-theoretical partial function defined by the expression. In particular, a higher order functional language need to be extended only with a *request* operator and a constant for explicit undefinition to serve as a data parallel language of this kind.

A less radical use of the formalism is to give formal semantics to the side-effect free parts of existing data parallel languages. For instance, it seems that the "pure" data parallel parts of HPF, i.e., array operations and `FORALL` statements, can be treated in this way. A formal semantics for these constructs could aid the construction of correct compilers, and algebraic laws like those in Section 5.5 can be of use in the design of optimizing transformations. Furthermore, a mathematical formalism like ours can guide the design of the data parallel primitives in new production languages.

We are currently experimenting with functional languages based directly on the formalism. A first experiment has been carried out, where a simple interpreter was implemented [27]. This interpreter implements a *request* operator which accepts functions over tuples of integers and performs a dynamic search for a finite "hyperrectangle" (i.e., a predicate of the form $\lambda i_1 \cdots i_n.(l_1 \leq i_1 \leq u_1 \wedge \cdots \wedge l_n \leq i_n \leq u_n)$) approximating the index set. Next on the agenda is to define and implement a more serious kernel language, with a rich set of basic operations, which essentially is a functional language with the minimal extensions necessary to support our formalism. This language is intended as an intermediate form, designed to be easy to analyze and transform, and the intention is to experiment with more user-friendly syntaxes which can be readily translated into this format, concurrently with the development of compiler technology for the kernel language.

Our formalism was originally intended to model the data parallel primitives occurring mainly in scientific computing. But as the work has progressed, we have discovered some other interesting possibilities as well:

- In the expression $f\backslash b$, the predicate b could well be given in a non-committed-choice logic programming language like Prolog. Under a *request*, the logic

program would then compute its set of solutions, put these in a data structure, and then f would be applied to each of these to create the resulting table. Thus, explicit restriction can be used as a semantically clean interface beween functional and logic languages.
- Data fields could be used to model relational databases, and operations on data fields could thus be used to express data base queries.
- Data fields over two- and three-dimensional spaces can be used for *symbolic representation of images* and *solid modelling*. Images can be defined by pixel-valued data fields. Operations such as translations, rotations, scaling etc. are readily expressible through function composition. *Printing* an image simply means requesting the pixel values in a number of points. *Symbolically represented movies* are images dependent on a also a time coordinate: *viewing* a movie then amounts to requesting the images defined at certain times. *The geometry of solids* can be defined by explicit restrictions in three dimensions. Data fields ranging over these restrictions can hold attributes such as density, colour, stress, etc., and these can be used for computing. *Viewing a 3-D scenery* means requesting a 2-D plane of pixels from some viewpoint, computed according to the projected 3-D scenery of data fields.

Exploring these applications are additional research topics of great interest.

8 Acknowledgements

Most of this text is based on joint work with Per Hammarlund [28, 29]. Extent analysis is joint work with Jean–François Collard [41]. I would also like to thank Karl-Filip Faxén, Joacim Halén, Olof Johansson and Claes Thornberg, who have all contributed in one way or another. Support has been given by The Swedish Research Council for Engineering Sciences (TFR), grants 91–333 and 94–109. Part of the underlying work was done while the author was Invited Professor at Ecole Normale Supérieure de Lyon.

References

1. S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 137–149. ACM, June 1990.
2. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, August 1978.
3. J. Backus, J. H. Williams, and E. L. Wimmers. An introduction to the programming language FL. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, chapter 9, pages 219–247. Addison-Wesley, Reading, MA, 1989.
4. H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1981.

5. R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, chapter 11, pages 287–307. Addison-Wesley, Reading, MA, 1989.
6. R. S. Bird. Constructive functional programming. In M. Broy, editor, *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.
7. G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie-Mellon University, Sept. 1995.
8. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.
9. G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, Feb. 1990.
10. L. Bougé. The data-parallel programming model: A semantic perspective. This Volume.
11. W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. Programming Languages. McGraw-Hill, 1990.
12. D. Cann. Retire Fortran? A debate rekindled. *Comm. ACM*, 35(3):81–89, Aug. 1992.
13. B. Carpentieri and Z. G. Mou. A formal approach to Divide-and-Conquer parallel computation and its applications on the Connection Machine. In *Proceedings of the First Italian Conference: Algorithms and Complexity*, pages 13–44. World Scientific Publ., Oct. 1990.
14. B. Carpentieri and Z. G. Mou. Compile-time transformations and optimization of parallel Divide-and-Conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, Oct. 1991.
15. S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1993.
16. M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *J. Parallel Distrib. Comput.*, pages 461–491, 1986.
17. M. C. Chen, Y.-I. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7, pages 255–308. Addison-Wesley, 1991.
18. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, Jan. 1977.
19. J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *Proc. PARLE'93*, Volume 694 of *Lecture Notes in Comput. Sci.*, pages 146–160, Athens, June 1993. Springer-Verlag.
20. J.-M. Delosme and I. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems: an illustration of a methodology for the construction of systolic architectures in VLSI. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 37–46, Bristol, UK, 1987. Adam Hilger.
21. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976.

22. K. Ekanadham. A perspective on Id. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. Addison-Wesley, 1991.
23. P. Feautrier. Automatic parallelization in the polytope model. This Volume.
24. 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.
25. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
26. N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* Thèse de 3eme cycle, Univ. de Grenoble, Mar. 1979.
27. J. Halén, P. Hammarlund, and B. Lisper. An experimental implementation of a highly abstract model of data parallel programming. In preparation, 1996.
28. 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.
29. P. Hammarlund and B. Lisper. Purely functional data parallel programming. Submitted, 1995.
30. High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0.* Technical report CRC-TR-92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.
31. J. M. Hill. The AIM is laziness in a data-parallel language. In K. Hammond and J. T. O'Donnell, editors, *Glasgow functional programming workshop.* Springer-Verlag, 1993.
32. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, Dec. 1986.
33. C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Inform.*, 24:595–632, 1987.
34. P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.
35. P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992.
36. K. E. Iverson. *A Programming Language.* Wiley, London, 1962.
37. S. Y. Kung. VLSI array processors. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 7–24, Bristol, UK, 1987. Adam Hilger.
38. H. Le Verge, V. Van Dongen, and D. K. Wilde. Loop nest synthesis using the polyhedral library. Research Report 840, IRISA, May 1994.
39. B. Lisper. *Synthesis of Synchronous Systems by Static Scheduling in Space-Time*, volume 362 of *Lecture Notes in Comput. Sci.* Springer-Verlag, Berlin, May 1989. (Ph. D. thesis).
40. B. Lisper. Synthesis of time-optimal systolic arrays with cells with inner structure. *J. Parallel Distrib. Comput.*, 10(2):182–187, Oct. 1990.
41. B. Lisper and J.-F. Collard. Extent analysis of data fields. In B. Le Charlier, editor, *Proc. International Symposium on Static Analysis*, Vol. 864 of *Lecture Notes in Comput. Sci.*, pages 208–222, Namur, Belgium, Sept. 1994. Springer-Verlag.
42. Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.
43. B. McKenney and B. Szymanski. Generating parallel code for SIMD machines. *ACM Letters on Programming Languages and Systems*, 1(1):59–73, Mar. 1992.
44. D. I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comput.*, C-31:1121–1126, Oct. 1982.

45. P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Int. Symp. on Comput. Arch.*, pages 208–214, June 1984.
46. P. Quinton, S. Rajopadhye, and D. Wilde. Derivation of data parallel code from a functional program. In *9th International Parallel Processing Symposium*, pages 1–15, 1995.
47. P. Quinton, S. Rajopadhye, and D. Wilde. Deriving imperative code from functional programs. In *7th Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995.
48. S. V. Rajopadye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, 1990.
49. S. K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proc. IEEE*, 76(3):259–269, Mar. 1988.
50. S. K. Skedzielewski. Sisal. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 4, pages 105–157. Addison-Wesley, 1991.
51. D. B. Skillicorn. Architecture-independent parallel computation. *Computer*, 23(12):38–50, Dec. 1990.
52. 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.
53. B. K. Szymanski. EPL–parallel programming with recurrent equations. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 3, pages 51–104. Addison-Wesley, 1991.
54. Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in C**, 6.1 edition, 1991.
55. Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in *Lisp*, 6.1 edition, 1991.
56. D. Turner. Functional programming and communicating processes. In *Proc. PARLE'87 vol. 2*, Volume 259 of *Lecture Notes in Comput. Sci.*, pages 54–74, Berlin, 1987. Springer-Verlag.
57. E. Violard. Data-parallelism versus functional programming: The contribution of PEI. Technical report, ICPS, Université Luis Pasteur, Strasbourg, Jan. 1995.
58. E. Violard and G.-R. Perrin. PEI: A language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.
59. J. A. Yang and Y. Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structures*, Montreal, Canada, June/July 1992.

This article was processed using the LaTeX macro package with LLNCS style