The Data Field Model

Björn Lisper and Per Hammarlund
Mälardalen University
and
Royal Institute of Technology (KTH)
Stockholm, Sweden

June 19, 2001

Abstract

Indexed data structures are prevalent in many programming applications. Collection-oriented languages provide means to operate directly on these structures, rather than having to loop or recurse through them. This style of programming will often yield clear and succinct programs. However, these programming languages will often provide only a limited choice of indexed data types and primitives, and the exact semantics of these primitives will sometimes vary with the data type and language.

In this paper we develop a unifying semantical model for indexed data structures. The purpose is to support the construction of abstract data types and language features for such structures from first principles, such that they are largely generic over many kinds of data structures. The use of these abstract data types can make programs and their semantics less dependent of the actual data structure. This makes programs more portable across different architectures and facilitates the early design phase. The model is a generalisation of arrays, which we call *data fields*: these are functions with explicit information about their domains. This information can be conventional array bounds but it could also define other shapes, for instance sparse.

Data fields can be interpreted as partial functions, and we define a meta-language for partial functions. In this language we define abstract versions of collection-oriented operations, and we show a number of identities for them. This theory is used to guide the design of data fields and their operations so they correspond closely to the more abstract notion of partial functions. We define φ-*abstraction*, a lambda-like syntax for defining data fields in a shape-independent manner, and prove a theorem which relates φ-abstraction and λ-abstraction semantically. We also define a small data field language whose semantics is given by formal data fields, and give examples of data field programming for parallel algorithms with arrays and sparse structures, database quering and computing, and specification of symbolic drawings.

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 values. Simple examples are homogenous lists and arrays, but the concept also includes more complex structures such as trees, graphs, nested sequences, hash tables, data parallel entities, and relational databases. 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 are

when solving partial differential equations, where the index is closely related to a physical coordinate, in image and signal processing, and in linear algebra.

The traditional way to compute with indexed data structures is to explicitly loop or recurse through them. It has since long been recognized, however, that a programming model which provides operations directly on the data structures can be very convenient. This model is an instance of *collection-oriented programming* [58]. The classical example is APL [16], which provides arrays and a rich set of operations on them.

Indexed data structures are very important in high performance computing. The data parallel programming model [25] is a collection-oriented paradigm for explicit parallelism, originally for SIMD architectures where distributed entities, like arrays indexed by processor coordinates, are manipulated in parallel. Many historical data parallel languages, like C* and *Lisp for the Connection Machine [64, 65], modeled the underlying machine closely: for instance, they typically demanded arrays to have the same dimensions as the processor array. This made it easier to generate efficient code for a particular machine but was less flexible from a programming point of view. Modern array languages, like Fortran 90 [7], High Performance Fortran (HPF) [24] and Sisal [17, 59], provide a less machine-dependent programming model where array dimensions need not be related to machine size. This makes them more flexible with regard to programming, but it becomes harder to compile them into efficient code. Still, they are better in that regard than traditional languages since collection-oriented programs tend to expose more of the inherent parallelism than programs with explicit loops or recursions. This also facilitates the kind of reordering transformations which are useful for improving the instruction level parallelism and cache hit ratio.

In general, modern computers have very complex performance characteristics. As a consequence, an important task when programming for high performance is to find elaborate data structures which provide the right tradeoffs between parallelism, locality, and memory consumption. This is particularly crucial when considering the possibility of sparse algorithms and representations, and dense/sparse hybrid solutions. Thus, there is a need to support the rapid prototyping of such structures and associated algorithms: on an abstract level, close to the problem formulation and thus portable, but also the transition to concrete implementations for different architectures. We believe this process can be facilitated by a programming model which aids the parameterization of programs with respect to data structures.

In other situations, the ability for concise modeling is more important than performance. The matrix language MATLAB [53], for instance, is widely used in education and engineering. Also the scripting language Perl provides collection-oriented features, like association arrays and advanced string operations. These languages are not known to have fast implementations, their popularity stems from the fact that they enhance the productivity of programmers for certain applications. We believe that also application-specific collection-oriented languages benefit from well-designed underlying models for the collections, which are designed according to common principles.

The data field model is an attempt to provide such principles for indexed data structures, through a common semantical framework for such structures. Another objective is to make the collection-oriented paradigm applicable to new problems by supporting a very generic programming model. The approach is to first consider the more abstract view of indexed structures as partial functions, and then add the explicit extra information about their domains which is necessary to perform all desired operations on them, while not making any undue restrictions. We call this extra information "bounds", and they are essentially set representations. This approach leads naturally to an axiomatic definition of bounds, where some operations with certain properties are postulated. We thus view bounds as an abstract data

type.

Partial functions can be defined through λ-abstraction and we define a similar syntax, called *φ-abstraction*, for data fields. Array languages often have convenient constructs to define arrays whose bounds are given implicitly. An important purpose of φ-abstraction is to provide a semantics for such constructs, and to aid the generalisation of them to other data structures. In particular, we target sparse structures.

A data field defined by φ-abstraction has its bounds implicitly given such that they approximate the domain of the corresponding partial function safely. Again, since there are tradeoffs between efficient implementations and exactness of domain approximation, we use an axiomatic approach where only certain properties of the rewrite system are prescribed. Thus, we actually define a class of φ-calculi adhering to the axioms. These calculi are given as higher order rewrite systems. We prove confluence and demonstrate that the well-known leftmost-outermost reduction strategy is normalizing for all φ-calculi in this class. We then formulate a theorem which relates the semantics of a φ-expression with the semantics of the corresponding λ-expression, and we prove it for an increasingly specialized suite of φ-calculi where rewrite rules are successively added to make the calculation of bounds more precise. We also outline an alternative class of φ-calculi, which define the kind of computation of implicit bounds traditionally found in array languages, and indicate how to define general mutable data fields which can be updated in-place.

What are the benefits of the data field approach? In Sipelstein's and Blelloch's survey of collection-oriented languages [58], a number of common collection-oriented operations are defined and a taxonomy is introduced. All these operations can be expressed in a kernel language consisting of a minimal functional language enriched with bounds and their operations, a function constructing data fields from functions and bounds, and φ-abstraction. Sipelstein and Blelloch furthermore identified certain semantical ambiguities for some of these operations. If these operations are defined as outlined above, then the ambiguities are resolved in a natural fashion.

Programming languages can provide data fields and φ-abstraction, and will then enable a highly generic style of programming where a change of data collection representation will require only a minimal change in the code. Some codes are fully "bounds-generic" – an example is found in Sect. 8.4. The theorem about semantical correspondence between φ- and λ-expressions ensures that the semantics of φ-expressions in the code will not change in a certain sense (to be made precise in Sect. 7) when the underlying data structure is changed. We believe this property is especially useful in the early design phase of collection-oriented programs, where different representations and algorithms are tried out.

φ-abstraction, as defined here, enables a lazy style of programming where infinite data fields make sense. The advantage of lazy evaluation and infinite data structures from a software-engineering perspective is well known. Two particular applications in collection-oriented programming are the use of infinite constant data fields which adapt to the right shape when used in a certain context, and predicates which act as "masks" over finite data fields. See Sect. 2.

In [45], we defined a highly generic framework for *extent analysis* of data structures, using the more abstract view of indexed data structures as partial functions. We have also defined and implemented "Data Field Haskell", a dialect of Haskell where the arrays are replaced with an instance of data fields (essentially the sparse/dense arrays of Sect. 7.5), according to the framework defined here [27, 30, 29, 47].

The rest of this paper is organized as follows. In Sect. 2 we give a taxonomy of collection-oriented operations and identify the major syntactical styles for these. Sect. 3 provides a first, informal definition of data fields and a small motivating example in the form of a simple data field language and some programs. In Sec-

tions 4-6 we develop a small metalanguage for partial functions and describe how almost all collection-oriented operations can be conveniently expressed in this language. We also give a number of identities for these operations. Section 7 provides the formal definitions of data fields and all related concepts, some results are proved, and an example of a possible instance of data fields is developed. A substantial part is devoted to φ-abstraction and how their bounds can be computed. In Sect. 8 we give some larger examples how our small data field language can be used to express a variety of collection-oriented algorithms in a generic and convenient way. Sect. 9 gives an account for related work. In Sect. 10, finally, we wrap up and give some directions for future research.

An early presentation of the more abstract data-structures-as-partial-functions model, given here in Sections 5-6, is found in [23]. In [43], a tutorial over this model is given. A short, preliminary account for the data field model as presented here is given in [44].

2 Operations on indexed data structures

Which operations on indexed data structures are there, then, and how are they expressed syntatically? One can distinguish six major groups of operations and three syntactical styles. All operations considered in [58] fall into some of these groups or can be expressed through operations from these.

Elementwise applied operations (*apply-to-each* in [58]) apply a "scalar" operation f to every element a in a data structure A. That is, the resulting data structure will contain the elements $f(a)$, where a belongs to A. The canonical example is the `map` operation on lists.

A common extension is to allow elementwise applied operations taking several arguments. `map` provides this on lists if there are `zip_`n functions available which create lists of n-tuples from n lists. Data parallel and array languages usually provide direct syntactical support. The simplest syntax is to introduce a syntactically distinct elementwise applied operation for each "scalar" operation. It is often more convenient to *overload* the scalar operation. In Fortran 90,

```
X+Y
```

denotes the elementwise addition of `X` and `Y` if these are arrays. An operation that can be overloaded in this way is called an *elemental intrinsic*. This assumes a typing of `X` and `Y`, such that the overloading can be resolved.

A third, comprehension-like kind of notation "quantifies" over a given range of indices, in order to explicitly mention each of the individual elements of the resulting structure. The parallel `for` construct in Sisal is an example: adding `X` and `Y` elementwise for indices `1` to `n` can be expressed as

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

The `FORALL` statement [1] in HPF is very related.

The exact semantics of elementwise applied operations varies, in particular when they take several arguments with different extents. For operations on lists as above, the semantics of `zip_`n decides the semantics of the elementwise applied operation (typically, `zip_`n yields a list as long as the shortest argument). Another common solution is to require *conformance* of the operands, which for one-dimensional arrays means that they must have the same length. The semantics is that the arrays are aligned and then added elementwise. But what should the *index range* of the result be? In the imperative language Fortran 90, the range of a right-hand side array expression in an array assignment is given by the range of the left-hand side. But

what if the expression occurs in some other environment, as in a purely functional language?

A second group of operations reorder data structures (*permute operations* in [58]). A common operation is "parallel read" (*inverse permute* in [58]) from a data structure A, where, for each index i within some range, $A(source(i))$ is selected. Here, *source* is some function from indices to indices, which possibly is defined by another indexed data structure. Parallel read can be expressed in the same three ways as above, in HPF, for instance, through `FORALL`:

```
FORALL (I=1:N) B(I) = A(SOURCE(I))
```

where `SOURCE` is an array. HPF and Fortran 90 also support the overloaded syntax `B = A(SOURCE)`. This is precisely indexing by arrays like in APL, which thus can be seen as parallel read where the range of the result is the range of the index array.

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. In languages with elemental intrinsics, the following syntax is often allowed:

```
A+17
```

where `A` is an array. The meaning of this expression is an array, with the same range as `A`, whose element for each index `i` is `A(i)+17`. This can be seen as a two-step operation where first an array with the same range as `A`, filled with the value `17`, is created, and then these arrays are elementwise added. This automatic replication of a scalar into an array is sometimes called *promotion*. 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, or rows.

A fourth group of operations *select* parts of data structures. *Projections* are common in array languages: these select subarrays of lower dimension. For instance, in Fortran 90, `A(1,:)` refers to the first row of the matrix `A`. *Restriction* operations apply a boolean condition elementwise, as a "mask", in order to select a part of a data structure. In Fortran 90,

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

effectively sets every element of `A` to its absolute value. Also dimension-preserving subarray selection, like

```
A(I:J)
```

in Fortran 90, which selects the subarray of `A` ranging from `I` to `J`, can be seen as a restriction, as well as the range specification in a HPF `FORALL` statement. Projection and restriction are also common operations on relational databases.

A common restriction operation for lists is a `filter` function which yields a list of all elements in a list where a predicate is true. There is a subtle difference in the semantics: for array operations, selected elements usually retain their indices. The filter operation, on the other hand, typically produces a "compressed" list where the selected elements have their positions changed.

Domain operations (a case of *information operations* in [58]) return some kind of information about the domain of the data structure. Examples are the length of a list, or the bounds of an array. A domain operation which is often implicit in the structure is an *ordering* of the elements. Domain information is needed to build catenation operations – an important group of operations which can be derived from the kinds of operations listed here.

Reduction operations, finally, compute some value as a function of the elements of a data structure. Usually, the function is composed of some repeatedly applied binary operation. If the operation is associative, the reduction can be implemented

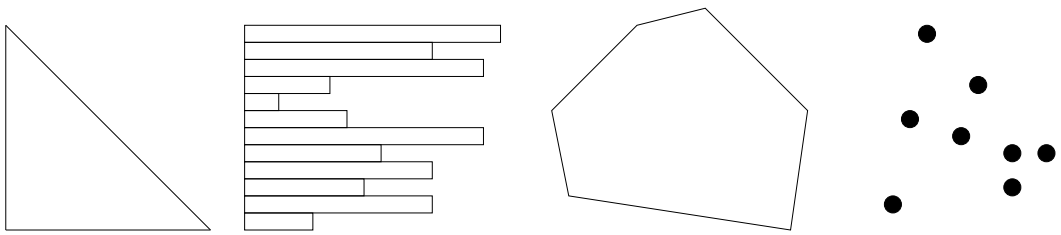

Figure 1: Some possible generalised bounds.

in parallel, and if it is commutative the execution order is even less restricted. Examples of reduction operations are the `foldl` and `foldr` operations on lists.

Reduction is often thought of as an operation over a multiset of values (e.g., summing them). But unless the binary operation is associative and commutative the elements must be ordered if the reduction is to be well-defined. For lists and one-dimensional arrays, there are natural orderings. The situation is less clear for multidimensional arrays and sparse structures.

3 Data Fields, Informally

The term "data fields" is borrowed from Crystal [10]. Our data fields are pairs (f, b), where f is a function and the "bound" b is a set representation. We denote the set defined by b by $\{\{b\}\}$, and the corresponding predicate by $[\![b]\!]_{bool}$. $\{\{b\}\}$ contains indices from which arguments to f may be drawn. $(f, b)\,!\,x$ denotes application of (f, b) to x. Operationally, this means "if $x \in \{\{b\}\}$ then return $f(x)$ else return an out-of-bounds error value". This defines an interpretation of (f, b) as a partial, hyperstrict function whose domain is contained in $\{\{b\}\}$. The canonical example is the array: then b is a tuple of index bounds and $[\![b]\!]_{bool}$ is a conjunction of linear inequalitites defining a "hyperrectangle" in the index space of the array. But we can also allow other bounds which yield less restricted "array shapes", such as triangular shapes, nested shapes, general convex polyhedra, and finite sparse structures, see Fig. 1. We can allow more general associations, indexed by other data types than tuples of integers. We can even have data fields with bounds defining infinite sets, although operations that require all the defined elements in the data field will not be applicable to such fields.

We define data fields to be hyperstrict when seen as functions since memoised data fields seem important (in conventional models for high performance computing memoised structures are the norm). Although nonstrict lookup procedures are possible [32], hyperstrict lookup is easier to implement and seems conceptually simpler.

The out-of-bounds error value, which we denote by "$*$", has algebraic properties similar to the divergent element $\perp$ (see [46] for details), and sometimes we will even identify them (as in Haskell [31]) although they represent quite different behaviours. When they are not identified we will sometimes consider a test "is_*" which returns *true* for $*$.

Clearly, the bounds are central to the data field concept. To keep the concept of bounds generic our approach is to view bounds as an abstract data type, where certain operations and properties are postulated, rather than constructing the bounds explicitly. Operations on data fields can then be defined using the abstract operations on bounds. Languages could either have predefined instances of bounds, or provide means for programmers to define their own bounds and operations on them. In the former case, operations on bounds and data fields can be given specialised,

efficient implementations. In the latter case it is convenient if the host language has some kind of class system which can be used to overload the operations.

The postulates for bounds are roughly the following (exact formulations are given in Sect. 7):

- Every bound has an interpretation as a predicate (or set).
- There are binary operations $\sqcap$, $\sqcup$ on bounds that correspond to (possible over-approximations of) the intersection and union operations on the sets defined by the bounds. (They are not to be confused with the domain-theoretic g.l.b. and l.u.b.)
- There are two bounds *all* and *nothing* that represent the universal and empty set, respectively.
- A bound is either *finite* or *infinite*, depending on whether the set defined by it is surely finite or possibly infinite.
- For every bound b defining a finite set $\{\{b\}\}$, $size(b)$ yields the size of $\{\{b\}\}$ and $enum(b)$ is a function enumerating the elements in $\{\{b\}\}$.

Why do we postulate these operations? The classification of bounds into infinite or finite is needed since certain operations on data fields, like reduction, are well-defined only for finite data fields. Applying a data field to an argument requires a test that the argument is within the bounds: thus, the need to interpret bounds as predicates. Size and enumeration are needed for finite data fields to make reduction and other iterative operations over them well-defined, since we then must know in general which elements to reduce over and in which order. $\sqcap$ and $\sqcup$ are used in the propagation of bounds which occurs when reducing φ-expressions as defined in Sect. 7.1. *all* and *nothing*, finally, can also appear as a result of this reduction.

Two interesting derived operations are *reduction* (or fold) "$red_{\mathcal{D}}$" of a data field with a binary operation, and *explicit restriction* "$\downarrow$" of a data field with a bound. The former is a fold of the elements in the data field, in the order given by the enumeration of its bound, and the latter returns a data field which is the first argument with its bound intersected with the second argument. Exact definitions are found in Sect. 7.

Our particular requirements on bounds stem from their intended use in collection-oriented programming. Other applications may have other requirements. In set-based program analysis, for instance, enumerations are not important while a test for equality becomes essential. Relational databases (which also can be seen as sets) have a third set of operations, distinct from the other two.

The postulated operations on bounds have been selected to only require that elements of a finite bound can be ordered. An important extension, which we make in Sect. 7.3, is to define *product bounds* and their properties. If b_1 and b_2 are bounds, for instance, then (b_1, b_2) is a two-dimensional bound where b_1 constrains the first dimension and b_2 the second. The predicate, finiteness, size, enumeration, "intersection", and "union" of product bounds are all derived from the corresponding operations on the components, as defined in Sect. 7.3. Product bounds can thus be used to define multidimensional data fields. However, multidimensional bounds can also be non-product bounds, and we exemplify in Sect. 7.6.

The canonical example of bounds is conventional array bounds. In the one-dimensional case, these are pairs (l, u) of integers. These bounds are apparently finite. We have $[\![(l,u)]\!]_{bool} = \lambda x.l \leq x \leq u$, $size(l,u) = \max(u \perp l + 1, 0)$, $enum(l,u) = \lambda x.(x \perp l)$, $(l,u) \sqcap (l',u') = (\max(l,l'), \min(u,u'))$, and $(l,u) \sqcup (l',u') = (\min(l,l'), \max(u,u'))$. See Fig. 2. Note that $\sqcup$ may overapproximate the union of the sets given by its operands. Multidimensional array bounds can be constructed

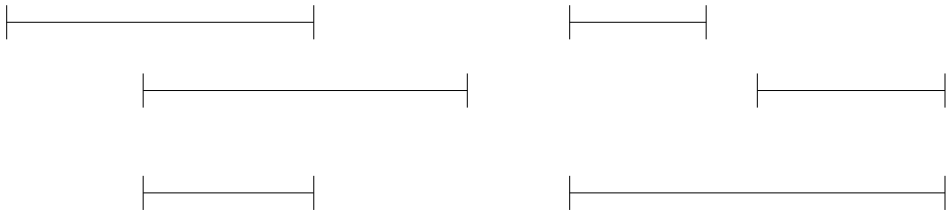

Figure 2: $\sqcap$ and $\sqcup$ on array bounds.

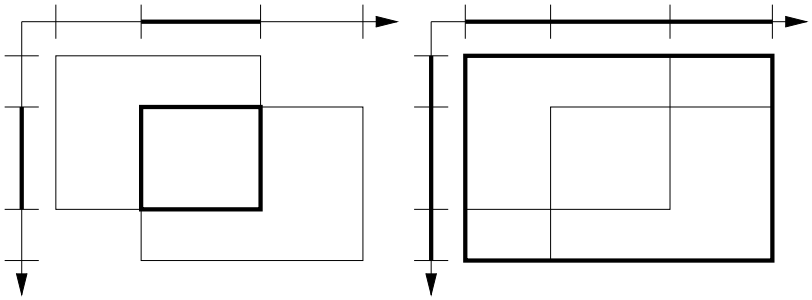

Figure 3: $\sqcap$ and $\sqcup$ for multidimensional array bounds.

as products of one-dimensional bounds. See Fig. 3 for an illustration. Sect. 7.5 contains a thoroughly worked example of more general sparse/dense array bounds and their operations.

In Sect. 7.1 we introduce φ-abstraction, which provides a formal "forall" kind of syntax for defining data fields. The term $\varphi x.t$ can be read "for all x (where t is defined), t", and it defines a data field $(\lambda x.t, b)$ where b is derived from bounds of data fields occurring in t. Exact definitions are given in Sect. 7 through rewrite systems.

Derivation of bounds in some "natural" way is common in array languages, and it can relieve the programmer from tedious specifications of bounds. Three kinds of operations that often provide this derivation are elementwise application of scalar operator, selection and projection on higher-dimensional arrays, and indirect indexing. Some array languages also provide shift or translation operations with this facility, and selection operations with possible non-unit stride. The semantics of φ-abstraction is designed to provide derivations of bounds for these operations. Since the operations have different scope we actually define three different φ-calculi. Elementwise application and indirect indexing can be defined regardless of the kind of data field, and our first rewrite system defines how bounds are "propagated" for these operations. Selection and projection operations are specific for higher-dimensional data fields, and our second rewrite system, which extends the first, defines how bounds are derived for these. Translation and selection with stride, finally, is defined only for data fields indexed by (tuples of) integers, and our third rewrite system adds rewrite rules that define how bounds are derived for these operations.

The bound for a term $\varphi x.t$ is designed to provide an approximation of the domain for the corresponding partial function $\lambda x.t$. We believe this is a particularly "natural" way to define bounds. The domain of $\lambda x.t$ can be defined in terms of set operations on domains of partial functions occurring in t, and the bounds for $\varphi x.t$ are then defined through the corresponding, postulated operations on bounds. Thus, our way to derive bounds does not require any *a priori* choice of data structure. For elementwise applied strict operations this approach leads naturally to the "implicit

intersection rule" known from FIDIL [57].

Our particular rewrite systems only provide some of many possible ways to define the derivation of bounds. There is a tradeoff between how "tightly" a bound can approximate the domain of a partial function and how complex the derivation is. Our rewrite systems are designed to be easy to modify for other tradeoffs. Furthermore, there are situations where the requirement of conformance for operands of elementwise applied operations can be the most natural choice (for instance, if we have a matrix algebra then elementwise addition of differently sized matrices could be considered a kind of type error). In Sect. 7.9 we outline a slightly modified rewrite system which enforces the conformance requirement rather than the implicit intersection rule.

3.1 A First Example

It is possible to define a small but powerful and generic core language for data fields from a small, conventional "host language" extended with bounds, data fields, and φ-abstraction. On top of this language, the syntax can be proliferated to meet different needs in special applications. As an example, we define a minimal higher order language extended with data fields from a subset of the *sparse/dense arrays* defined in Sect. 7.5. The language has types

$$\tau ::= \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid \tau_1 \to \tau_2 \mid \texttt{Df}\ \tau_1\ \tau_2 \mid \texttt{Bnds}\ \tau_1$$

and expressions

$$\begin{aligned} t \quad ::= \quad & n \mid \texttt{True} \mid \texttt{False} \mid x \mid t_1\ \ aop\ \ t_2 \mid \texttt{not}\ t_1 \mid t_1\ \ bop\ \ t_2 \mid t_1\ \ rop\ \ t_2 \\ & \mid \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \backslash x\texttt{->}\ t_1 \mid t_1\ t_2 \\ & \mid \texttt{forall}\ x\texttt{->}\ t_1 \mid t_1\ \texttt{at}\ t_2 \mid t_1 \texttt{!} t_2 \mid \texttt{fold} \\ & \mid \texttt{oub} \mid \texttt{isoub} \mid \texttt{in} \mid \texttt{size} \mid \texttt{enum} \mid \texttt{all} \mid \texttt{nothing} \mid t_1 \texttt{:} t_2 \end{aligned}$$

Here, n ranges over integer constants, *aop* over arithmetical operators, *bop* over boolean connectives, and *rop* over relational operators. x ranges over identifiers. We assume all identifiers have a given typing. Identifiers which are not introduced locally are defined by a global declaration.

This is an explicitly typed, higher order variation of the language REC in [70], extended with data field primitives. The typing rules for the extensions are straightforward and we omit them here. The data-field-free part of the language can be given a denotational semantics in a standard way, as a function mapping from terms and identifier-binding environments to elements in cpo's, with semantical entities expressed in the metalanguage in Sect. 4. The extensions can be given a semantics in the following way, using informally introduced entities that will be formally defined in Sections 4, and 7:

$$\begin{aligned}
[\![\texttt{forall}\ x\texttt{->}\ t]\!]\rho &= [\![\varphi x.(t\rho|_x)]\!]_{\mathcal{D}(\tau_1,\tau_2)} & [\![\texttt{in}]\!]\rho &= \lambda b.[\![b]\!]_{bool} \\
[\![t_1\ \texttt{at}\ t_2]\!]\rho &= [\![t_1]\!]\rho \downarrow [\![t_1]\!]\rho & [\![\texttt{size}]\!]\rho &= \mathit{size} \\
[\![t_1 \texttt{!} t_2]\!]\rho &= [\![t_1]\!]\rho\,!\,[\![t_2]\!]\rho & [\![\texttt{enum}]\!]\rho &= \mathit{enum} \\
[\![\texttt{fold}]\!]\rho &= \mathit{red}_{\mathcal{D}} & [\![t_1 \texttt{:} t_2]\!]\rho &= ([\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![\texttt{oub}]\!]\rho &= * & [\![\texttt{all}]\!]\rho &= \mathit{all} \\
[\![\texttt{isoub}]\!]\rho &= \mathit{is}_* & [\![\texttt{nothing}]\!]\rho &= \mathit{nothing}
\end{aligned}$$

In the semantics for `forall`, the type of `forall` x `->` t is assumed to be $\texttt{Df}\,\tau_1\,\tau_2$. The φ-expression in the right-hand side is a syntactic object, whose semantics $[\![\cdot]\!]_{\mathcal{D}(\tau_1,\tau_2)}$ is defined in Section 7.1 through a confluent rewrite system. $\rho|_x$ stands for ρ with the binding for x removed. $t\rho|_x$ is the term that results when $\rho|_x$ is applied as a substitution to t.

We deliberately define only a minimal number of primitives for constructing bounds at this point. "`:`" constructs a dense array bound from two integers. We also allow predicates $\tau \to$ `Bool` and data fields of type `Df` τ `Bool` to be (infinite) bounds of type `Bnds` τ. Finally, there are finite sparse bounds. We give no explicit way to construct them, but they result from the intersection of a finite bound with a predicate or boolean data field.

We now introduce Fortran 90 array style overloading of arithmetical operators as a kind of syntactic sugar, which can be removed by a type-directed source-to-source transformation "$\rightsquigarrow$":

$$\begin{array}{rcl} t_1, t_2\texttt{: Df } \tau \texttt{ Int} & \Longrightarrow & t_1 \ \mathit{aop}\ t_2 \ \rightsquigarrow \ \texttt{forall x->(}t_1\texttt{!x } \mathit{aop}\ t_2\texttt{!x)} \\ t_1\texttt{: Int}, t_2\texttt{: Df } \tau \texttt{ Int} & \Longrightarrow & t_1 \ \mathit{aop}\ t_2 \ \rightsquigarrow \ \texttt{forall x->(}t_1\ \mathit{aop}\ t_2\texttt{!x)} \\ t_1\texttt{: Df } \tau \texttt{ Int}, t_2\texttt{: Int} & \Longrightarrow & t_1 \ \mathit{aop}\ t_2 \ \rightsquigarrow \ \texttt{forall x->(}t_1\texttt{!x } \mathit{aop}\ t_2\texttt{)} \end{array}$$

We define this kind of overloading similarly for the other kinds of operators in the language.

As a simple example of a data field definition we now define a function for computing histograms over data fields. First, some Haskell-style definitions for convenience:

```
sum d = fold (+) d 0
b2i x = if x then 1 else 0
```

The histogram over a data field in general can now be expressed as the function

```
hist d = forall x-> sum forall y-> (b2i (x = d!y))
```

How does this definition work? the `x` in the outer `forall` ranges over the indices of the result. This is an infinite data field, not surprisingly since the domain of a histogram depends on the range of the field being histogrammed over, and this range cannot be known á priori in general. The data field defined by the inner `forall` will have the same bound as `d`. This is since `d!y` occurs in a strict position in the body. The sum thus ranges over this bound, and the net result is that for any `x` the number of occurrences of elements in `d` equal to `x` is computed. (Note the similarity between the idiom "`sum forall y->`" and "$\sum_y$".)

If we know something about the range of `d`, for instance that elements of `d` must lie in the range $1 \ldots n$, then we can restrict the bound of the histogram accordingly:

```
(hist d) at 1:n
```

The bound of this data field is `1:n` and it is thus finite. This demonstrates the lazy nature of data fields. It is furthermore a "dense" data field, which possibly contains many zeroes, for instance if the size of `d` is much less than `n`. We may therefore want to define a *sparse* histogram, which is defined only in the points where it is nonzero. We define a general "data field sparsifier" for this purpose:

```
sparse d = d at d /= 0
```

We can now write

```
(sparse (hist d)) at 1:n
```

to obtain a sparse histogram over the nonzero values of `d` in the range $1 \ldots n$.

Note the declarative and generic nature of these definitions. The only place where the index type of `d` is "given away" is in the restriction with the bound `1:n`. Apart from that, `d` could be indexed by any valid index type for data fields, and the definitions could thus be reused for any kind of data field.

4 Preliminaries

4.1 A Metalanguage for Partial Functions

We now define a small metalanguage for partial functions. (Essentially this is a variation of the metalanguage for continuous functions in [70].) Since we want to be able to embed our concepts into various host languages we do not specify all the details of the language completely; rather, we give a language scheme. We consider the following kind of cpo's:

- *Basic cpo's* which are flat cpo's denoted by constant symbols, in particular the flat cpo of booleans *bool* and the flat cpo of integers *int*,
- Products of cpo's, constructed with $\times$, lifted sums of cpo's, constructed with $+$, and cpo's of continuous functions constructed with $[\rightarrow]$,
- Recursively defined cpo's given by equations $D = \mathcal{F}(D)$, where $\mathcal{F}(D)$ is built out of the cpo variable D, cpo constants, and the cpo operations $\times$, $+$, and $\rightarrow$.

These cpo's correspond to recursive types defined in the usual way. We assume that elements in basic cpo's can be compared for equality. We define *Eq-cpo's* as above, but excluding the function cpo operation $\rightarrow$ in the definitions. Eq-cpo's correspond to the "Eq-types" of ML. Now, we define the following language scheme for defining elements in these cpo's:

- Symbols D ranging over (pointed) cpo's.
- For every cpo D a constant "$\bot_D$" denoting the bottom element (usually, we will just write "$\bot$"), and a constant "$*_D$" (usually written "$*$") denoting a distinguished error value. $*$ is an isolated, maximal element such that only $\bot$ lies below it.
- For every Eq-cpo D a predicate $is_{*D} \in [D \rightarrow bool]$, usually written "$is_*$", defined by:

$$\begin{aligned} is_{*D}(*_D) &= true \\ is_{*D}(\bot_D) &= \bot_D \\ is_{*D}(x) &= false, \quad x \notin \{*_D, \bot_D\} \end{aligned}$$

- The following constructors: *tupling* $(,\ldots,)$ to construct elements of product cpo's, and *injections* in_i to construct elements of sum cpo's.
- A number of n-ary function symbols denoting n-ary operations over flat cpo's, which are strict (in all arguments) when no argument equals $*$. If some argument equals $*$, then the function value must be either $*$ or $\bot$.
- Variables typed with the cpo they range over (e.g. $f \in [D \rightarrow E]$). (We use set membership notation to suggest the intended interpretation.) We will omit the typing when it is not necessary.
- For every cpo D a conditional $if_D \in [bool \times D \times D \rightarrow D]$, usually written "*if*", defined by:

$$\begin{aligned} if_D(true, x, y) &= x \\ if_D(false, x, y) &= y \\ if_D(\bot, x, y) &= \bot \\ if_D(*_{bool}, x, y) &= *_D \end{aligned}$$

- λ-abstraction and function application.
- A least fixed point operator μ. We will often give recursive definitions rather than using μ explicitly.

We use the n-ary notation $\lambda(x_1, \ldots, x_n).t$, where $x_i \in D_i$, to denote a function in $[D_1 \times \cdots \times D_n \to E]$. In principle, we have a typed λ-calculus with constants, where the cpo inclusions $t \in D$ are typing judgements. It is clear (see [70]) that every well-formed term of type D in this language has a direct interpretation as an element in D: in particular, function-typed terms denote continuous functions. We will often, especially in equations, let terms in the language denote their interpretations directly.

On the other hand, if we remove the constants $\bot_D$, then we have a syntax for a simple, higher-order functional language. In Sect. 5 we will use it as a convenient syntax to define collection-oriented operations on partial functions. In Sections 7.1 and 7.2 we will consider terms in the metalanguage as terms rather than elements of cpo's, and consider rewrite semantics rather than denotational semantics. The connection between rewite and denotational semantics is well known [3, 12]. Higher order languages can be given rewrite semantics based on Klop's Combinatory Reduction Systems (CRS), see Appendix B, and in Appendix A we define a CRS M which gives an alternative rewrite semantics for the metalanguage. In Sect. 7.1 we will successively extend the metalanguage with constructions for data fields and bounds, including φ-abstraction, and we extend M to cover the extended languages.

4.2 Hyperstrictness

Hyperstrictness was first defined by Turner [67]. His definition was informal. We give more stringent definitions of hyperstrictness and related concepts below.

Definition 1 *For any element d in an Eq-cpo D and for any element d' in an appropriate cpo, we define the relation "in" by: d' in d iff:*

- $d' = d$,
- $d = (d_1, d_2)$, *and* d' *in* d_i *for some* i, *or*
- $d = \in_i (d_1)$ *for some* i, *and* d' *in* d_1.

Definition 2 *For any cpo D, $\overline{D} \subseteq D$ is defined by: $d \in \overline{D}$ iff $d \sqsubseteq d' \implies d = d'$, and $d = \bigsqcup_{i=0}^{\infty} d_i \implies \exists i.d = d_i$.*

We call the elements in $\overline{D}$ *finite maximal* elements in D. Clearly, for elements d in Eq-cpo's we have that d is finite maximal iff d is of finite size and it does not hold that $\bot$ *in* d.

Definition 3 *For any $f \in [D \to D']$, where D is an Eq-cpo, $\overline{f} \in D \to D'$ is defined by:*

$$\overline{f}(x) = \begin{cases} f(x), & x \in \overline{D} \wedge \neg(* \text{ in } x), \\ *, & x \in \overline{D} \wedge * \text{ in } x, \\ \bot & \text{otherwise.} \end{cases}$$

f is hyperstrict *if $f = \overline{f}$.*

It is straightforward to verify that $\overline{f}$ is continuous whenever f is. We have $\overline{\overline{f}} = \overline{f}$. Operationally, the evaluation of a hyperstrict function terminates only if its argument can be fully computed in finite time.

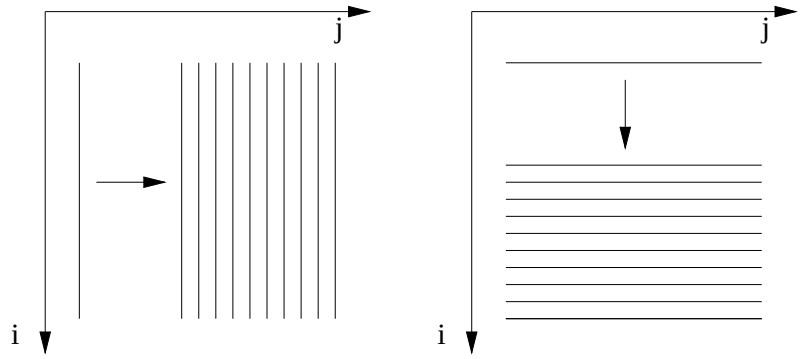

Figure 4: "replicate as columns" $\lambda(i,j).f(i)$ and "replicate as rows" $\lambda(i,j).f(j)$.

5 Collection-Oriented Operations on Partial Functions

In this section we define higher order operations in the metalanguage of Sect. 4, which yield abstract versions of all the collection-operations from Sect. 2 except the domain operations.

Elementwise application of an n-ary operation g on the partial functions $f_1, \ldots, f_n$ is a kind of function composition: $\lambda x.g(f_1(x), \ldots, f_n(x))$. It will often be convenient to use "elemental intrinsics style overloading" of g and write $g(f_1, \ldots, f_n)$. 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 $f(g)$). This models both "indexing with arrays" (when g is a partial function) and communication schemes such a shifts, permutations, broadcasts, etc.

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 for each possible value for x. This provides an exact notation for replicating an array along some axis into an array of higher dimension. For instance, a "one-dimensional" function f can be replicated into the "matrices" $\lambda(i,j).f(i)$ ("replicate as columns") and $\lambda(i,j).f(j)$ ("replicate as rows"). See Fig. 4.

λ-abstraction also provides a convenient notation for *projection*. For instance, if f represents a matrix, then $\lambda i.f(i,1)$ represents the first row of f.

Explicit restriction of a partial function f w.r.t. the predicate b is defined viz.: $f \setminus b = \lambda x.\mathit{if}(b(x), f(x), *)$. The following result is easy to prove. It will become useful in Sect. 7.

Proposition 1 *For any $f \in [D \to D']$ and $b \in [D \to \mathit{bool}]$ it holds that $\overline{f \setminus b} = \overline{f} \setminus \overline{b}$.*

Reduction with respect to a binary operation op over a partial function f is performed over all elements of the domain of f taken in some particular order, provided that this domain is finite. It can be given a simple recursive definition in the metalanguage. We need to provide explicit domain information: an integer n which gives the size of the domain, and an *enumeration* i of the domain (a function from $\{0, \ldots, n \perp 1\}$ to the domain of f). We assume that op has a left identity element e such that $op(e,x) = x$ for all x. Then, reduction "*red*" is defined by:

$$\begin{aligned} \mathit{red}(op, f, e, i, 0) &= e \\ \mathit{red}(op, f, e, i, n) &= \mathit{let}\ r = \mathit{red}(op, f, e, i, n \perp 1)\ \mathit{in} \\ &\qquad \mathit{if}(\mathit{is}_*(f(i(n \perp 1))), r, op(r, f(i(n \perp 1)))), \quad n > 0 \end{aligned}$$

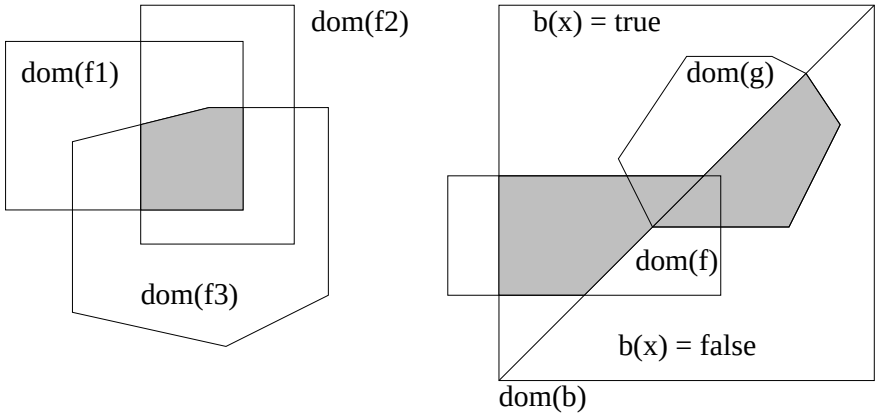

Figure 5: The domain of $g(f_1, f_2, f_3)$ and of $if(b, f, g)$. ($dom(f)$ stands for $\{ x \mid f(x) \neq \bot \}$.)

This definition "filters out" points x where $f(x) = *$. Thus, it is appropriate to use in situations where n and i stem from an overapproximating bound for a data field. Indeed, we will use it when defining reduction for data fields in Sect. 7. The definition is not parallel. But if *op* is associative, then it is very easy to show that this definition is equivalent to a balanced recursion with $O(\log n)$ recursion depth.

red can be used to define other common reduction-like operation on partial functions, such as *scan* (parallel prefix), *segmented reduction*, and *segmented scan*. See [46].

6 Some Simple Identities

We now give a number of identities for the explicit restriction operator defined in Sect. 5. Clearly, they can be used for program optimizations. The aforementioned extent analysis [45] is based on these rules: this analysis tries to find the "extent" of a partial function, i.e., its domain, at compile-time. The laws have also inspired how bounds are computed for φ-abstraction, as defined in Sections 7.1 and 7.7.

All the results below follow more or less immediately, usually by a simple case analysis. "$\wedge$" below refers to the non-strict version of conjunction, for which $false \wedge \bot = false$, extended to handle $*$ in the following way:

$$\begin{aligned} * \wedge x &= * \\ true \wedge * &= * \\ false \wedge * &= false \\ \bot \wedge * &= \bot \end{aligned}$$

Note that $\wedge$ can be defined through the *if*-conditional in the metalanguage. We use elemental intrinsics syntax throughout.

Proposition 2 *(Flattening of nested restrictions)* $(f \setminus b) \setminus b' = f \setminus (b' \wedge b)$.

Proposition 3 *(Communication of restriction)* $(f \setminus b)(g) = f(g) \setminus b(g)$, *and if* g *has a left inverse* g^{-1}, *then* $f(g) \setminus b = (f \setminus b(g^{-1}))(g)$.

6.1 Elementwise Application

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 operation:

Proposition 4 *(Elementwise application) For any i, $g(f_1, \ldots, f_i, \ldots, f_n) \setminus b = g(f_1, \ldots, f_i \setminus b, \ldots, f_n) \setminus b$.*

This identity explains why the *live-domain analysis* of FIDIL [57] is valid.

Proposition 5 and its two corollaries below hold under the condition that $*$ *and* $\perp$ *are identified*, i.e., that the equation $\perp = *$ is valid. Since $*$ represents an error condition detectable in finite time and $\perp$ represents divergence, this means that the different sides of the equality possibly may have different termination properties. Thus, the results can be used if we don't care about the distinction between error and nontermination (similar to *lenient semantics* [15]), or in a situation where we know that the functions involved will always terminate, (like, say, if they are memoised data structures). In [46] the situation when $*$ and $\perp$ are kept distinct is investigated in depth.

Proposition 5 *(Elementwise application of strict function) 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 1 *If g is strict in all arguments, then $g(f_1 \setminus b_1, \ldots, f_n \setminus b_n) = g(f_1, \ldots, f_n) \setminus (b_1 \wedge \ldots \wedge b_n)$.*

Finally, for the elementwise applied conditional, we have the following identity:

Proposition 6 *$\mathit{if}\,(b \setminus b', f \setminus b_f, g \setminus b_g) = \mathit{if}\,(b, f, g) \setminus b' \wedge ((b \wedge b_f) \vee ((\neg b) \wedge b_g))$.*

(Negation is extended to $*$ by $\neg * = *$.) The validity of this identity requires that the identities $\mathit{false} \vee x = x \vee \mathit{false} = x$ holds. Thus, $\vee$ must be extended to handle $\perp$ and $*$ so these identities still hold. This is for instance the case if it is extended to be evaluable in a left-to-right fashion, similarly to the previously extended $\wedge$. Corollary 1 and Proposition 6 are illustrated in Fig. 5.

7 Data Fields

In Sect. 3 we defined data fields informally as pairs of functions and bounds. We now give formal definitions. First we define exactly what we require from bounds, and we then proceed to define data fields.

Definition 4 *Let α be an Eq-cpo. The cpo $\mathcal{B}(\alpha)$ is a* cpo of bounds for α *if the following operations, with the properties below, are defined:*

finite	$\in$	$[\mathcal{B}(\alpha) \to \mathit{bool}]$	*test for finiteness*
$[\![\cdot]\!]_{\mathit{bool}}$	$\in$	$[\mathcal{B}(\alpha) \to [\alpha \to \mathit{bool}]]$	*interpretation as predicate*
enum	$\in$	$[\mathcal{B}(\alpha) \to [\mathit{int} \to \alpha]]$	*enumeration*
size	$\in$	$[\mathcal{B}(\alpha) \to \mathit{int}]$	*size*
$\sqcap$	$\in$	$[\mathcal{B}(\alpha) \times \mathcal{B}(\alpha) \to \mathcal{B}(\alpha)]$	*intersection of bounds*
$\sqcup$	$\in$	$[\mathcal{B}(\alpha) \times \mathcal{B}(\alpha) \to \mathcal{B}(\alpha)]$	*union of bounds*
all_α	$\in$	$\overline{\mathcal{B}(\alpha)}$	*universal bound*
$\mathit{nothing}_\alpha$	$\in$	$\overline{\mathcal{B}(\alpha)}$	*empty bound*

The following properties should hold:

- *If $b \in \overline{\mathcal{B}(\alpha)}$ and $\mathit{finite}(b)$, then $\mathit{size}(b) \geq 0$. If furthermore $\mathit{size}(b) > 0$, then $\mathit{enum}(b)|_{\{0,\ldots,\mathit{size}(b)-1\}}$ is a bijection from $\{0, \ldots, \mathit{size}(b) \perp 1\}$ to $\{\{b\}\} = \{\, x \mid [\![b]\!]_{\mathit{bool}}(x) = \mathit{true} \,\}$.*
- *If $b, b' \in \overline{\mathcal{B}(\alpha)}$, then $b \sqcap b' \in \overline{\mathcal{B}(\alpha)}$, $b \sqcup b' \in \overline{\mathcal{B}(\alpha)}$, $\{\{b\}\} \cap \{\{b'\}\} \subseteq \{\{b \sqcap b'\}\}$, and $\{\{b\}\} \cup \{\{b'\}\} \subseteq \{\{b \sqcup b'\}\}$.*

- $[\![all_\alpha]\!]_{bool} = \lambda x.true$, $[\![nothing_\alpha]\!]_{bool} = \lambda x.false$, *and* $size(nothing_\alpha) = 0$.

We will usually drop the index α when it is clear from the context. We say that b is *finite* if $finite(b) = true$, otherwise *infinite*. If $size(b) = 0$ then b is *empty*. Thus, *nothing* is empty, but other bounds may also be empty. Definition 4 actually defines a class of continuous algebras [12, 20] but we will not use this fact here.

Definition 5 *If* $[\alpha \to \beta]$ *is a cpo of continuous functions, and if* $\mathcal{B}(\alpha)$ *is a cpo of bounds for* α*, then* $\mathcal{D}(\alpha, \beta) = [\alpha \to \beta] \times \mathcal{B}(\alpha)$ *is a* cpo of data fields from α to β.

In the sequel symbols f stand for functions, d for data fields, and b for bounds. The following proposition is needed to prove Lemma 2 in Sect. 7.2.

Proposition 7 $b \sqsubseteq b' \implies \{\{b\}\} \subseteq \{\{b'\}\}$.

Proof. If $b \sqsubseteq b'$ then, by monotonicity of $[\![\,\cdot\,]\!]_{bool}$, follows that $[\![b]\!]_{bool}(x) = true \implies [\![b']\!]_{bool}(x) = true$, which yields the result. ■

We now define some derived operations on data fields:

Definition 6 *The following functions are defined by the following equations.* Interpretation of data field as function, $[\![\,\cdot\,]\!]_{\alpha\to\beta} \in [\mathcal{D}(\alpha, \beta) \to [\alpha \to \beta]]$*:*

$$\begin{aligned} [\![(f, b)]\!]_{\alpha\to\beta} &= \overline{f \setminus [\![b]\!]_{bool}} \\ [\![\bot_{\mathcal{D}(\alpha,\beta)}]\!]_{\alpha\to\beta} &= \bot_{\alpha\to\beta} \end{aligned}$$

Data field application, $! \in [\mathcal{D}(\alpha, \beta) \times \alpha \to \beta]$*:*

$$d\,!\,x = [\![d]\!]_{\alpha\to\beta}\,x$$

Explicit restriction of data field, $\downarrow \in [\mathcal{D}(\alpha, \beta) \times \mathcal{B}(\alpha) \to \mathcal{D}(\alpha, \beta)]$*:*

$$\begin{aligned} (f, b) \downarrow b' &= (f, b' \sqcap b) \\ \bot_{\mathcal{D}(\alpha,\beta)} \downarrow b &= \bot_{\mathcal{D}(\alpha,\beta)} \end{aligned}$$

Reduction of data field, $red_{\mathcal{D}} \in [(\beta \times \gamma \to \gamma) \times \mathcal{D}(\alpha, \beta) \times \gamma \to \gamma]$*:*

$$red_{\mathcal{D}}(op, (f, b), e) = red(op, f, e, enum(b), size(b))$$

We will drop the index $\alpha \to \beta$ for the interpretation and write $[\![d]\!]$ when it is clear that d is a data field and its type is not important. Reduction of data fields is defined directly from the reduction over partial functions in Sect. 5. Explicit restriction of data fields is modeled after the restriction "\" on partial functions, and its definition is inspired by Proposition 2.

7.1 φ-abstraction

We now define φ-abstraction formally. This is a syntax with bound variables, exactly analogous with λ-abstraction, which defines data fields. Thus, we extend the metalanguage of Sect. 4 with terms formed according to the following rule:

$$\frac{x \in \alpha \quad \alpha \text{ Eq-type} \quad t \in \beta}{\varphi x.t \in \mathcal{D}(\alpha, \beta)}$$

In the rest of this section and the first part of Sect. 7.2 we consider terms in the meta-language as terms rather than elements of cpo's, and names of cpo's as types. The reason for this change of view is that we will define the semantics of φ-abstraction by rewrite systems. We define the semantics relative to some host language, which at a minimum contains operations on data fields and bounds as specified in Definitions 4 – 6. In order to stay fully within this kind of semantics we will assume that also the semantics for the host language is given by some rewrite system R in this part of the paper. We furthermore assume that if $t \leftrightarrow_R^* t'$, where $\leftrightarrow_R^*$ is the convertibility relation generated by R, then t and t' have the same denotational semantics. (This is a basic soundness property for rewrite system semantics.)

Actually we define three different semantics, which yield an increasingly precise computation of bounds: first a "basic" semantics which is independent of the type of indices; then, in Sect. 7.3, an extension to define and handle multidimensional data fields; and then, in Sect. 7.7, a further extension to handle scalings and offsets of array-like data fields. The semantics are given as higher order rewrite systems (Combinatory Reduction Systems, or CRS: see Appendix B) $\Phi_i(R)$, $i = 0, 1, 2$. Thus, $\Phi_i(R) \cup R$ gives the rewrite semantics for the host language extended with φ-abstraction of "version i".

In particular, the host language could be the metalanguage of Sect. 4 extended with the operations defined earlier in Sect. 7. In Appendix A an orthogonal CRS M is defined for the metalanguage, and we can assume a CRS I for the extension. Then, $\Phi_i(M \cup I) \cup M \cup I$ gives rewrite semantics for the metalanguage extended with data fields, bounds, their basic operations, and φ-abstraction.

We assume that $M \cup I$ is orthogonal. We do not give I explicitly (its exact definition will depend on the kind of bounds, and how the postulated operations on them are defined). Note, however, that any definition $f(x) = t$ can be directed into a rewrite rule $f(x) \to t$. This applies directly to the definition of derived operations in Definition 6 and to the instances of the postulated operations on sparse/dense array bounds given in Sect. 7.5. For these bounds $M \cup I$ will indeed be orthogonal.

Definition 7 *For any cpo γ, let T_γ be the set of terms of type γ, V_γ the set of variables of type γ, and V the set of variables of any type. B is a* bounds-computing function for R *if, for some Eq-cpo α and cpo β, it is a partial function $T_\beta \times V_\alpha \times 2^V \to T_{\mathcal{B}(\alpha)}$ such that $B(t, x, Y)$ is defined if and only if:*

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

Now, each $\Phi_i(R)$ is defined by a rule scheme

$$\varphi x.t \to (\lambda x.t, B_i(t, x, \emptyset)) \tag{1}$$

where B_i is a bounds-computing function for R, which defines one rule for each t such that $B_i(t, x, \emptyset)$ is defined. It is possible to give a more operational definition of each B_i through a rewrite system R_i: $B_i(t, x, Y)$ is then seen as a term in itself, rather than as a metaterm whose meaning is given by the function B_i. If R_i is orthogonal, mutually orthogonal with $\Phi_i(R) \cup R$, and terminating, then $R_i \cup \Phi_i(R) \cup R$ is orthogonal, and, in a normalising reduction strategy, the evaluation of $B_i(t, x, Y)$ can be carried out a soon as t fulfils the conditions in Definition 7. The rewrite semantics given by R_i is then consistent with the partial function view, where each right-hand side in (1) is "precomputed". As will be seen shortly, $B_i(t, x, Y)$ is defined over the structure of t so the properties above are natural.

However, defining R_i would require the modelling of explicit representations of sets of variables and we prefer to keep the definitions of the B_i free from such details. A definition of the reduction rules (1) in formal CRS syntax is given in Appendix C.

We have chosen to give a rewrite semantics to φ-abstraction for a number of reasons. One reason is to highlight the similarities and differences to β-reduction in the λ-calculus. Another reason is that languages which have a rewrite semantics are referentially transparent w.r.t. the convertibility relation of the rewrite system. Furthermore, it is well known how to relate rewrite semantics to both denotational semantics and operational semantics. We discuss the operational semantics of φ-abstraction in Sect. 7.2. For the denotational semantics, note that $B_i(t, x, Y)$ is hyperstrict in t since it is defined only when t is a normal form. Thus, the computation of $\varphi x.t$ diverges unless t has a normal form, and we define

$$[\![\varphi x.t]\!]_{\mathcal{D}(\alpha,\beta)} = \begin{cases} (\lambda x.t', B_i(t', x, \emptyset)), & \text{if } t' \text{ is the normal form for } t \\ \bot, & \text{if } t \text{ has no normal form} \end{cases}$$

(In Sect. 7.2 we show that $\Phi_i(R) \cup R$ is confluent under mild conditions on R: thus, normal forms are unique, so $[\![\ \cdot\ ]\!]_{\mathcal{D}(\alpha,\beta)}$ is well-defined.) Finally, we define $[\![\varphi x.t]\!]$, i.e. the function defined by $\varphi x.t$, as $[\![[\![\varphi x.t]\!]_{\mathcal{D}(\alpha,\beta)}]\!]_{\alpha\to\beta}$.

We now proceed to define the different $B_i(t, x, Y)$. For all i we define $B_i(t, x, Y)$ by cases over the possible syntactical forms of normal forms t. For simplicity, we assume that all bound variables are distinct:

$$B_i(c, x, Y) = \mathit{all} \quad (c \text{ closed nf} \neq *) \tag{2}$$

$$B_i(*, x, Y) = \mathit{nothing} \tag{3}$$

$$B_i(y, x, Y) = \mathit{all} \quad y \in \{x\} \cup Y \tag{4}$$

$$B_i(op(t_1, \ldots, t_m), x, Y) = B_i(t_1, x, Y) \sqcap \cdots \sqcap B_i(t_m, x, Y) \quad (op \text{ strict}) \tag{5}$$

$$B_i(\mathit{if}(t_1, t_2, t_3), x, Y) = B_i(t_1, x, Y) \sqcap (B_i(t_2, x, Y) \sqcup B_i(t_3, x, Y)) \tag{6}$$

$$B_i(\lambda y.t, x, Y) = B_i(t, x, \{y\} \cup Y) \tag{7}$$

$$B_i(\varphi y.t, x, Y) = B_i(t, x, \{y\} \cup Y) \tag{8}$$

$$B_i((f, b)\,!\,x, x, Y) = b, \quad FV(f, b) = \emptyset \tag{9}$$

$$B_i((f, b)\,!\,t, x, Y) = B_i(t, x, Y), \quad FV(f, b) = \emptyset, t \neq x \tag{10}$$

For cases not explicitly covered, where $B_i(t, x, Y)$ still should be defined, we assume a default definition

$$B_i(t, x, Y) = \mathit{all}. \tag{11}$$

Let us motivate these definitions informally: a more formal account will follow in Sect. 7.2. (2) is appropriate since $\varphi x.c$ should be defined for all x, unless $c = *$ in which case it should be nowhere defined as stated by (3). Also $\varphi x.x$ should be defined everywhere which partly motivates (4). (5) corresponds to the implicit intersection rule in FIDIL [57] and is motivated by Corollary 1. In (5) we require that *op* is *consistently extended* to from a strict function on a $*$-free cpo to a cpo where $*$ is added, which basically means that it should handle $*$ in the same way as $\bot$ (see [46] for details). An important function which is not a consistently extended operation is the test is_*. (6) is similarly motivated by Proposition 6: informally, it holds since $\varphi x.\mathit{if}(t_1, t_2, t_3)$ should be defined only when t_1 is defined and some of t_2, t_3 are. (7) and (8) are treated below. $\varphi x.(f, b)\,!\,x$ should be defined only for arguments in $\{\{b\}\}$, which motivates (9). (10), finally, is sound since (f, b) defines a hyperstrict function: therefore, $\varphi x.(f, b)\,!\,t$ should be defined only when t is.

In $B_i(t, x, Y)$, x is the variable bound by the φ for which the bound is calculated. Y is the set of variables bound by other constructs under the φ, as effectuated by (7) and (8). Clearly, the bound derived for a φ-expression must not be dependent on

any variables bound inside the expression. Thus, the only reasonable approximation for their contribution to the bound is *all*.

By similar reasons we demand that $FV(f, b) = \emptyset$ in (9) and (10). Consider, for instance, $\varphi x.\lambda y.(f, b(y))\,!\,x$. This is a data field of functions, where the bound of the applied data field depends on the function argument. $b(y)$ is thus a locally defined entity and should not affect the bound of the φ-expression. It is interesting to contrast this case with $\lambda y.\varphi x.(f, b(y))\,!\,x$, which is a function returning data fields. Here, $B_i((f, b(y))\,!\,x, x, \emptyset)$ is not defined, since $FV(b(y)) = \{y\} \not\subseteq \emptyset$. This means that the bound for $\varphi x.(f, b(y))\,!\,x$ cannot be computed until y is instantiated to some value c and $b(c)$ is computed.

Definition 8 *$B_0(t, x, Y)$ is defined by equations (2) – (11).*

7.2 Properties of φ-abstraction

We now prove some results about φ-abstraction and discuss its properties. First there are four propositions about the rewrite systems. These rely only on the form of the rule scheme (1) and the fact that the B_i are bounds-computing functions for R: thus, they are valid for all rewrite systems $\Phi_i(R)$. The proofs are found in Appendix D.

Proposition 8 *$\Phi_i(R)$ is orthogonal.*

Proposition 9 *If R is left-linear and if no left-hand sides of any rules in R have any subterms of the form $\varphi x.t$, then $\Phi_i(R)$ and R are mutually orthogonal.*

Proposition 10 *If R in addition is confluent, then $\Phi_i(R) \cup R$ is confluent.*

Proposition 11 *Any φ-subterm of a closed $\Phi_i(R) \cup R$-nf must contain a variable bound by some other abstraction mechanism than φ.*

Why are these results interesting? Confluence guarantees uniqueness of normal forms, which means that every φ-term has a unique meaning. Orthogonality simplifies confluence proofs and also makes it possible to use standard results about reduction strategies, see below. A reduction strategy directly yields an operational semantics for evaluating φ-terms which is consistent with the rewrite semantics. Proposition 11, finally, implies that no φ-terms will remain after the evaluation of closed φ-terms, unless it is in the body of a remaining λ-abstraction (where it will, eventually, become reduced when the λ-abstraction is applied and thus its formal argument is instantiated).

What about reduction strategies for $\Phi_i(R) \cup R$ (and thus operational semantics for the language defined by this rewrite system)? Since the left-hand sides of all $\Phi_i(R)$ are closed, they are trivially left-normal [38]. If R in addition is orthogonal, left-normal, and satisfies the condition in Proposition 9, then each $\Phi_i(R) \cup R$ is orthogonal and left-normal. For rewrite systems with this property, the leftmost-outermost reduction strategy is normalising [37, 38].

However, every $\Phi_i(R)$ is defined by a rule scheme generating a possibly infinite number of rules. Thus, this generation of rules must be taken into account for the rewrite strategy. Given a term $\varphi x.t$ to evaluate, it must first be checked whether $B_i(t, x, \emptyset)$ is defined and then what it evaluates to. But it is easy to see that $B_i(t, x, \emptyset)$ is defined iff t is a $\Phi_i(R) \cup R$-nf such that $FV(t) \subseteq \{x\}$. Thus, an adequate operational semantics for a closed term $\varphi x.t$ is to first evaluate t to normal form t', and, if this succeeds, to compute $B_i(t', x, \emptyset)$ and reduce $\varphi x.t'$ accordingly. This fits into the left-normal evaluation strategy with the extension that $B_i(t', x, \emptyset)$ is evaluated before making the final reduction.

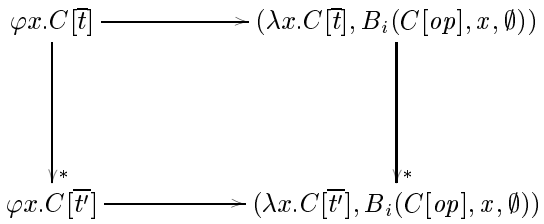

Figure 6: Confluence of reductions with explicitly hyperstrict terms. *op* stands for a hyperstrict operation such that $B_i(C[op], x, \emptyset)$ is defined.

Finally, note how the evaluation of $\varphi x.t\,!\,t'$ proceeds:

$$\varphi x.t\,!\,t' \to^* \varphi x.t''\,!\,t' \to (\lambda x.t'', b)\,!\,t' \to \overline{\lambda x.t'' \setminus [\![b]\!]_{bool}}\,t'$$

So there is no direct β-reduction for applied φ-abstractions. A data field must first be computed into its "data field normal form" of form (f, b) before it can be applied to its argument.

If $M \cup I$ is orthogonal, left-normal, and satisfies the conditions in Proposition 9, then $\Phi_i(M \cup I) \cup M \cup I$ is orthogonal and left-normal. Thus, an operational semantics for the metalanguage in Sect. 4, extended with data field constructs and φ-abstraction, can be derived as sketched above.

Our semantics for φ-expressions $\varphi x.t$ as a rewrite system given by (1), where t is a R-nf, has the advantage that it is reasonably straightforward and makes it easy to prove orthogonality. However, it will in general require "computing under the φ", with an open term, even when $\varphi x.t$ is closed. This is in contrast to the evaluation of function-typed terms, where evaluation typically is not done under a lambda. Some of this symbolic computing can always be done at compile-time, like application of a λ-term to an argument, but in general symbolic run-time computing may be required which can be costly.

A closer examination reveals two problematic cases: higher-order variables, and recursive definitions. An example of the first case is the term $\lambda g.\varphi x.d!g(x)$. Here, an implementation which is faithful to the rewrite semantics must delay the calculation of bounds for $\varphi x.d!g(x)$ until g is instantiated, and its value is symbolically evaluated with x as formal argument. Recursive functions applied to open arguments can give even worse problems: consider, for instance, $\varphi x.(\mu f.t\,x)$ where $t = \lambda y.if\,(y = 0, 0, y \cdot f(y \perp 1))$. $\mu f.t\,x$ has no normal form and thus the evaluation of $\varphi x.(\mu f.t\,x)$, which is necessitated by the rewrite semantics, will not even terminate!

A possible solution is to consider higher order variables and terms of the form $\mu f.t$, occurring under a φ, to be hyperstrict, i.e., a term t of one of these forms is seen as syntactic sugar for $\overline{t}$. Then, one can consider all terms of the form $\overline{t}$ as equivalent, w.r.t. B_i, to constant function symbols which stand for hyperstrict functions, and add the corresponding cases for B_i. For instance, (5) then gives rise to the new case

$$B_i(\overline{t}(t_1, \ldots, t_m), x, Y) = B_i(t_1, x, Y) \sqcap \cdots \sqcap B_i(t_m, x, Y),$$

even when t is not a normal form. A verification that $\Phi_i(R) \cup R$ still is confluent is outside the scope of this paper, but an informal motivation is given in Fig. 6.

We now show a theorem which relates φ-abstraction and λ-abstraction. It essentially states that if $*$ and $\perp$ are identified, then $[\![\varphi x.t]\!] = \lambda x.t$ when restricted to maximal elements. We show the theorem for the "basic" semantics for φ-abstraction given by Φ_0: later, we extend the theorem to the semantics given by Φ_1 and Φ_2. From now on we also revert, unless otherwise stated, to the view where terms in the metalanguage are considered elements in cpo's rather than syntactical terms.

Throughout the rest of Sect. 7.2, we will assume that $* = \bot$. In [46] we study the relationship between cpo's where $* = \bot$ and the corresponding cpo's where $\bot$ and $*$ are kept distinct: for a cpo D where these elements are equal we denote the corresponding cpo, where $*$ is distinguished, with D_*. Functions over D_* which distinguish $\bot$ and $*$ have no counterpart in D: in our metalanguage the only such function is is_*. Formally, the translation from D_* to D is given by a homomorphism ϕ which maps both $\bot_{D_*}$ and $*_{D_*}$ to $\bot_D$.

We will take an extensional view of functions and data fields. Thus, $f = *_{[\alpha\to\beta]}$ iff $f(x) = *_\beta$ for all $x \in \alpha$. Therefore, whenever $\bot_{[\alpha\to\beta]_*} \sqsubseteq f \sqsubseteq *_{[\alpha\to\beta]_*}$, it must hold that $\phi(f) = \bot_{[\alpha\to\beta]}$. Similarly, for data fields, we equate d with $*_{\mathcal{D}(\alpha,\beta)_*}$ whenever $[\![d]\!] = \overline{\lambda x.*_\beta}$. Again, whenever $\bot_{\mathcal{D}(\alpha,\beta)_*} \sqsubseteq d \sqsubseteq *_{\mathcal{D}(\alpha,\beta)_*}$, it holds that $\phi(d) = \bot_{\mathcal{D}(\alpha,\beta)}$. For instance, $\phi(\lambda x.*_\beta, b) = \bot_{\mathcal{D}(\alpha,\beta)}$ for any bound b.

We also assume, for the rest of this section, that the domain of bounds $\mathcal{B}(\alpha)$ under consideration is such that for any $b \in \mathcal{B}(\alpha)$ there is a maximal element $b' \in \mathcal{B}(\alpha)$ such that $b \sqsubseteq b'$. (By maximal we mean that $b' \sqsubseteq b'' \implies b' = b''$.) This is a technical condition which we need in order to prove Lemma 2, and it is fulfilled by all cpo's which appear in practice in denotational semantics.

Some notation: we define $res(f) = \{\, x \mid f(x) \neq \bot \,\}$, and $\{\{p\}\} = \{\, x \mid p(x) = \mathit{true} \,\}$ for all predicates p (similar to the notation $\{\{b\}\}$ for the set defined by the bound b). Let t be a term, Y a set of variables and v a type-preserving function from Y to values: then t^v denotes the resulting term when substituting $v(y)$ for y in t for all $y \in Y$.

The lemma below is useful in the proof of Lemma 2, and for proving the extended versions of this lemma which will appear in Sections 7.3 and 7.7.

Lemma 1 *$res(\lambda x.(f, b) \,!\, g(\vec{x})) \subseteq \{\{[\![b]\!]_{bool} \circ g\}\}$.*

Proof. See Appendix E. ■

The following lemma is a major stepping stone:

Lemma 2 *For all sets of variables Y, type-preserving mappings v from Y to values, variables x, and terms t such that $\lambda x.t^v$ is closed, there exists a bound b, which is maximal in $\mathcal{B}(\alpha)$, such that $res(\lambda x.t^v) \subseteq \{\{b\}\}$ and $B_0(t, x, Y) \sqsubseteq b$.*

Proof. See Appendix E. ■

Theorem 1 *For all terms t with normal form t' such that $B_0(t', x, \emptyset)$ is maximal, and for all finite maximal elements y, it holds that $\varphi x.t \,!\, y = \lambda x.t\, y$.*

Proof. See Appendix E. ■

Th. 1 is restricted to the case where $B_0(t', x, \emptyset)$ is maximal. It is possible to prove a theorem where this restriction is lifted, but then the property in Definition 4 that $\{\{b\}\} \cap \{\{b'\}\} \subseteq \{\{b \sqcap b'\}\}$ and $\{\{b\}\} \cup \{\{b'\}\} \subseteq \{\{b \sqcup b'\}\}$ must hold for *all* bounds, not just maximal bounds. It can be difficult to give a reasonable semantics for $\sqcap$ and $\sqcup$ for partially defined bounds so this property holds: thus, we have chosen the version above. It is really a matter of whether partially defined bounds should be considered valid set representations or not, and we have chosen not to consider them as such.

7.3 Multidimensional Data Fields

Multidimensional arrays are important in many applications. Thus, it is of great interest to provide adequate means to define multidimensional data fields. First, we identify a canonical way to define *product bounds* and their operations. These bounds generalise conventional multidimensional array bounds:

Definition 9 *Let $\mathcal{B}(\alpha_1), \ldots, \mathcal{B}(\alpha_k)$ be cpo's of bounds (for $k > 1$). Then $\mathcal{B}(\alpha_1) \times \cdots \times \mathcal{B}(\alpha_k)$ is* the cpo of canonical product bounds for $\alpha_1 \times \cdots \times \alpha_k$, *when its operations are given by:*

$$\begin{aligned}
\mathit{finite}(b_1, \ldots, b_k) &= \mathit{finite}(b_1) \wedge_X \cdots \wedge_X \mathit{finite}(b_k) \\
[\![(b_1, \ldots, b_k)]\!]_{bool}(x_1, \ldots, x_k) &= [\![b_1]\!]_{bool}(x_1) \wedge_X \cdots \wedge_X [\![b_k]\!]_{bool}(x_k) \\
\mathit{enum}((b_1, \ldots, b_k), n) &= (\mathit{enum}(b_1, n \bmod \mathit{size}(b_1)), \\
&\quad \mathit{enum}((b_2, \ldots, b_k), n \div \mathit{size}(b_1))), \\
&\quad \mathit{when}\ \mathit{size}(b_1) > 0 \\
\mathit{size}(b_1, \ldots, b_k) &= \mathit{size}(b_1) \cdot \ldots \cdot \mathit{size}(b_k) \\
(b_1, \ldots, b_k) \sqcap (b'_1, \ldots, b'_k) &= (b_1 \sqcap b'_1, \ldots, b_k \sqcap b'_k) \\
(b_1, \ldots, b_k) \sqcup (b'_1, \ldots, b'_k) &= (b_1 \sqcup b'_1, \ldots, b_k \sqcup b'_k) \\
\mathit{all}_{\alpha_1 \times \cdots \times \alpha_k} &= \mathit{all}_{\alpha_1} \times \cdots \times \mathit{all}_{\alpha_k} \\
\mathit{nothing}_{\alpha_1 \times \cdots \times \alpha_k} &= \mathit{nothing}_{\alpha_1} \times \cdots \times \mathit{nothing}_{\alpha_k}
\end{aligned}$$

"$\div$" is integer division.

Theorem 2 *A cpo of canonical product bounds for $\alpha_1 \times \cdots \times \alpha_k$ is a cpo of bounds for $\alpha_1 \times \cdots \times \alpha_k$.*

Proof. It is straightforward but tedious to verify that the operations indeed fulfil the requirements in Definition 4. We omit the details. ■

The notation "$\wedge_X$" in Definition 9 means that we do not specify the strictness properties fully; this operation just has to coincide with $\wedge$ on the fully defined truth values for the theorem to hold. Here and henceforth we will use the notation "op_X" when the strictness properties of *op* are left unspecified.

Note that *enum* orders $\{\{(b_1, \ldots, b_k)\}\}$ lexicographically with respect to the orders given by *enum* on $\{\{b_1\}\}, \ldots, \{\{b_k\}\}$, respectively. (This ordering is the same as the "column-major order" in which Fortran arrays are laid out in memory.) We will occasionally use the notation $\times_{i=1}^{k} b_i$ for the k-tuple $(b_1, \ldots, b_k)$.

Finally, note that bounds for $\alpha_1 \times \cdots \times \alpha_k$ need not be product bounds as given by Definition 9: for instance, it is possible to have multidimensional sparse bounds, which we consider in Sect. 7.6, or higher-dimensional non-rectangular polyhedral bounds.

We now introduce an extended φ-calculus for multidimensional bounds. Array languages often provide convenient constructs for, e.g., selecting rows and columns of matrices where the bounds of the resulting arrays are implicitly given by the bounds of the matrices. Our calculus generalises these constructs both to non-array-like data fields, and to other situations than selection and projection operations on arrays.

We use a pattern-matching syntax for φ-abstraction over tuples: $\varphi(x_1, \ldots, x_n).t$, which defines a data field in $\mathcal{D}(\alpha_1 \times \cdots \times \alpha_n, \beta)$ when $x_i \in \alpha_i$, for $i \in \{1, \ldots, n\}$, and $t \in \beta$. We use a similar syntax for λ-expressions. To avoid lenghty expressions we use the notation $\vec{t}$ for the tuple $(t_1, \ldots, t_n)$ when the arity n is understood from the context. In particular, $\vec{x}$ stands for $(x_1, \ldots, x_n)$, $\varphi\vec{x}.t$ for $\varphi(x_1, \ldots, x_n).t$ and $\lambda\vec{x}.t$ for $\lambda(x_1, \ldots, x_n).t$. Finally, we write $\vec{\alpha}$ for the n-ary tuple type $\alpha_1 \times \cdots \times \alpha_n$.

The CRS $\Phi_1(R)$ extends $\Phi_0(R)$ with more advanced derivation of multidimensional bounds. $\Phi_1(R)$ is defined through the bounds-computing function B_1, which is derived from B_0 by adding more explicit cases where the result is computed to something different than *all*. We also add cases where the second argument of B_1 is a tuple of variables, and $\Phi_1(R)$ will thus also contain rules $\varphi\vec{x}.t \to (\lambda\vec{x}.t, B_1(t, \vec{x}, \emptyset))$. For each of the equations (2) – (11) except (9) an equation with x replaced by $\vec{x}$ is added: we number these equations (2.1) – (11.1). If we introduce the convention that $\vec{x}$ can stand both for a tuple $(x_1, \ldots, x_n)$ and a single variable x, then (2) – (11) can be replaced by the new versions. (2.1) – (6.1) and (11.1) are straightforward to define. (7.1) and (8.1) are given by the following equations, for $i \in \{1, 2\}$ (by abuse of notation, we write $\vec{x}$ also for $\{x_1, \ldots, x_n\}$):

$$B_i(\lambda\vec{y}.t, \vec{x}, Y) = B_i(t, \vec{x}, \vec{y} \cup Y) \quad (7.1)$$
$$B_i(\varphi\vec{y}.t, \vec{x}, Y) = B_i(t, \vec{x}, \vec{y} \cup Y) \quad (8.1)$$

(10.1) will be defined later.

We do not introduce any new version of (9), since it will be subsumed by the rule introduced in this section. The new rule applies to cases where the components of $\vec{x}$ occur as individual arguments to data fields under the φ. This enables the use of φ-abstraction to express matrix operations such as projection, transposition and replication for general multidimensional data fields.

As an example, consider $\varphi(x_1, x_2, x_3).(f, (b_1, b_2, b_3, b_4))\ !\ (x_2, c, x_1, x_1)$. This is a three-dimensional data field, defined by selecting a two-dimensional subfield of the four-dimensional data field $(f, (b_1, b_2, b_3, b_4))$ which is then replicated in the x_3-direction. What should its bounds be? Obviously, x_2 should be constrained by b_1, and x_1 by both b_3 and b_4. x_3, on the other hand, should be left unconstrained. Furthermore it must be checked whether c belongs to $\{\{b_2\}\}$ or not: if not, then the resulting bound should be empty. We thus obtain the following expression for the bound: $\mathit{if}\,([\![b_2]\!]_{bool}(c), (b_3 \sqcap b_4, b_1, \mathit{all}), \mathit{nothing})$.

A variation of this example is $\varphi(x_1, x_2, x_3).\varphi y.(f, (b_1, b_2, b_3, b_4))\ !\ (x_2, y, x_1, x_1)$. This is a datafield of datafields. For the bound of the "outer" data field, it should *not* be checked whether y belongs to $\{\{b_2\}\}$, since y now is a variable bound inside the outer φ-abstraction rather than a constant. An appropriate bound is thus simply $(b_3 \sqcap b_4, b_1, \mathit{all})$. The bounds of an "inner" data field can be computed as soon as x_1, x_2, x_3 are instantiated (say, to c_1, c_2, c_3), and is then given by $\mathit{if}([\![b_1]\!]_{bool}(c_2) \wedge [\![b_3]\!]_{bool}(c_1) \wedge [\![b_4]\!]_{bool}(c_1), b_2, \mathit{nothing})$.

We need some formal notation for how the indices for the elements in $\vec{x}$ are mapped to positions in argument tuples[1]. Let $\vec{x}$ have arity n and $\vec{y}$ arity m, let I be a subset of $\{1, \ldots, m\}$, and let p be a partial function $\{1, \ldots, m\} \to \{1, \ldots, n\}$. Then $\vec{y}[p, \vec{x}]$ is an m-tuple defined by:

$$\vec{y}[p, \vec{x}]_j = \begin{cases} x_{p(j)}, & p(j) \text{ defined} \\ y_j, & \text{otherwise} \end{cases}$$

For instance, if $n = 3$, $m = 4$, $p(1) = 2$, and $p(3) = p(4) = 1$, then $\vec{y}[p, \vec{x}] = (x_2, y_1, x_1, x_1)$.

We now postulate, for every $b \in \mathcal{B}(\alpha_1 \times \cdots \times \alpha_m)$, partial function p: $\{1, \ldots, m\} \to \{1, \ldots, n\}$, $I \subseteq \{1, \ldots, m\}$, and $y_i \in \alpha_i$, $i \in \{1, \ldots, m\} \setminus \mathit{dom}(p) \setminus I$, a bound $\mathit{bproj}_{p,I}(b, \vec{y})$ in $\mathcal{B}(\alpha'_1 \times \cdots \times \alpha'_n)$ such that

$$\exists(\, y_i \mid i \in I \cup \mathit{dom}(p)\,).[\![b]\!]_{bool}(\vec{y}[p, \vec{x}]) = \mathit{true} \implies [\![\mathit{bproj}_{p,I}(b, \vec{y})]\!]_{bool}(\vec{x}) = \mathit{true} \quad (12)$$

[1] When $\vec{x}$ is a simple variable we consider it equivalent to a tuple with one element, and similarly for data field arguments which are not tuples.

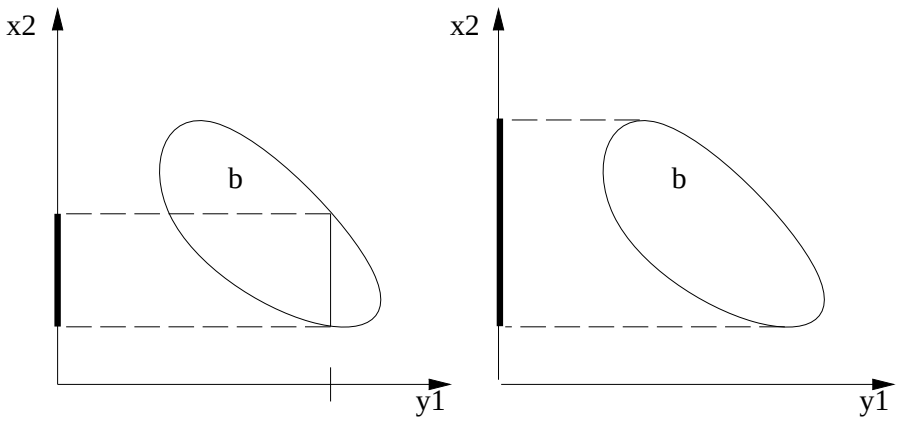

Figure 7: Selection and projection of b.

for all $\vec{x}$ in $\alpha'_1 \times \cdots \times \alpha'_n$. For instance, with the previous values of n, m, and p we obtain[2] $[\![bproj_{p,\emptyset}(b,\vec{y})]\!]_{bool}(\vec{x}) = [\![b]\!]_{bool}(x_2, y_1, x_1, x_1)$, and $[\![bproj_{p,\{2\}}(b,\vec{y})]\!]_{bool}(\vec{x}) = \exists y_1.[\![b]\!]_{bool}(x_2, y_1, x_1, x_1)$. We of course assume that types are respected, that is: $\alpha_i = \alpha'_{p(i)}$ for all i where $p(i)$ is defined.

bproj can be seen as a quite general selection/projection-operation on bounds. To see this, consider the following examples. Define p by $p(2) = 2$. We have $[\![bproj_{p,\emptyset}(b,\vec{y})]\!]_{bool}(\vec{x}) = [\![b]\!]_{bool}(y_1, x_2)$. This is a bound for x_2 which is a function of y_1. This can be seen as a *selection* of the "slice" of $[\![b]\!]_{bool}$ given by y_1. It is interesting to compare this with $[\![bproj_{p,\{1\}}(b,\vec{y})]\!]_{bool}(\vec{x}) = \exists y_1.[\![b]\!]_{bool}(y_1, x_2)$. This is rather a *projection* onto the x_2-axis of $\{\{b\}\}$. See Fig. 7. Other interesting examples are, with $p(1) = 2$, $p(2) = 1$: $[\![bproj_{p,\emptyset}(b,\vec{y})]\!]_{bool}(\vec{x}) = [\![b]\!]_{bool}(x_2, x_1)$ ("transpose" of b), and with $p(1) = 1$, $p(2) = 1$: $[\![bproj_{p,\emptyset}(b,\vec{y})]\!]_{bool}(\vec{x}) = [\![b]\!]_{bool}(x_1, x_1)$ ("main diagonal" of b).

We now define, whenever $FV(f,b) = \emptyset$,

$$B_i((f,b)\,!\,\vec{t}[p,\vec{x}],\vec{x},Y) = bproj_{p,I}(b,\vec{t}), \tag{13}$$

where: $\emptyset \subset FV(t_i) \subseteq Y$ for $i \in I$ and $FV(t_i) = \emptyset$ for $i \notin I$. Note that if $bproj_{p,I}(b,\vec{y}) = b$ when p is the identity function, which is in accordance with (12), then (13) subsumes (9) when $\vec{x}$ is a single variable. We can now give (10.1):

$$B_i((f,b)\,!\,\vec{t},\vec{x},Y) = B_i(\vec{t},\vec{x},Y), \quad FV(f,b) = \emptyset, \vec{t} \neq \vec{t'}[p,\vec{x}] \tag{10.1}$$

for all $\vec{t'}[p,\vec{x}]$ as defined in (13).

Definition 10 *For canonical product bounds, $bproj_{p,I}$ is defined by:*

$$bproj_{p,I}(\times_{i=1}^m b_i,\vec{y}) = if(\bigwedge_{i\in\{1,\ldots,m\}\setminus dom(p)\setminus I} [\![b_i]\!]_{bool}(y_i), \times_{j=1}^n(\sqcap_{p(j)=i} b_j), nothing)$$

Here, $\sqcap_{p(j)=i} b_j$ equals *all* if there is no j such that $p(j) = i$, and $\bigwedge_{i\in\emptyset} P_i$ is true for any propositions P_i.

Proposition 12 *$bproj_{p,I}(\times_{i=1}^m b_i,\vec{y})$ satisfies (12).*

Proof. Consider the left-hand side of (12) for $b = \times_{i=1}^m b_i$. By Definition 9 this expression equals $\exists(\, y_i \mid i \in I \cup dom(p)\,).\bigwedge_{i=1}^m [\![b_i]\!]_{bool}(\vec{y}[p,\vec{x}]_i)$. This evaluates to true iff:

[2] Assuming a "tight" *bproj* such that (12) is an equivalence.

1. For all $i \in dom(p)$, $[\![b_i]\!]_{bool}(x_{p(i)})$ is true,
2. For all $i \in \{1,\ldots,m\} \setminus dom(p) \setminus I$, $[\![b_i]\!]_{bool}(y_i)$ is true, and
3. For all $i \in I \setminus dom(p)$, there is a y_i such that $[\![b_i]\!]_{bool}(y_i)$ is true.

1 and 2 implies that $[\![bproj_{p,I}(\times_{i=1}^{m} b_i, \vec{y})]\!]_{bool}(\vec{x})$ is true. ■

Continuing the example above, with $\vec{y}[p,\vec{x}] = (x_2, y_1, x_1, x_1)$ and $I = \emptyset$, we obtain

$$bproj_{p,I}(\times_{i=1}^{m} b_i, \vec{y}) = if([\![b_2]\!]_{bool}(y_1), (b_3 \sqcap b_4, b_1, all), nothing).$$

We now formally define B_1 and extend Lemma 2 to cover B_1. Th. 1 then carries over directly to φ-expressions with semantics given by $\Phi_1(R) \cup R$, as can be seen by its proof in Appendix E.

Definition 11 *$B_1(t, X, Y)$ is defined by equations (2.1) – (8.1), (10.1), (11.1), and (13).*

Lemma 3 *For all sets of variables Y, type-preserving mappings v from Y to values, variable tuples $\vec{x}$, and terms t such that $\lambda\vec{x}.t^v$ is closed, there exists a bound b, which is maximal in $\mathcal{B}(\alpha)$, such that $res(\lambda\vec{x}.t^v) \subseteq \{\{b\}\}$ and $B_1(t,\vec{x},Y) \sqsubseteq b$.*

Proof. See Appendix E. ■

7.4 An Example: Multidimensional Language Features

We extend the simple data field language in Sect. 3.1 to handle multidimensional data fields. For that purpose we introduce *tuple types* $(\tau_1, \ldots, \tau_n)$, *tuple expressions* $(t_1, \ldots, t_n)$, and *pattern matching* on tuple arguments to lambda- and forall-abstraction: $\backslash(x_1, \ldots, x_n)\texttt{->}\, t$ and $\texttt{forall}(x_1, \ldots, x_n)\texttt{->}\, t$. We also define projections $\texttt{proj\_}1, \ldots, \texttt{proj\_}n$ which select elements of tuples. Tuples of type $(\texttt{Bnds}\ \tau_1, \ldots, \texttt{Bnds}\ \tau_n)$ are interpreted as canonical bounds over $(\tau_1, \ldots, \tau_n)$ (but we do not prohibit other multidimensional bounds). We obtain a language powerful enough to take full advantage of the propagation of bounds defined by Φ_1.

We now demonstrate how a common array notation for selection and projection can be generalised to data fields and put on top of our language. The syntax is

$$t\texttt{!}\,(u_1, \ldots, u_n)$$

where t: $\texttt{Df}\ (\tau_1, \ldots, \tau_n)\,\tau$ and each u_i is either an argument term of type τ_i, a bound of type $\texttt{Bnds}\ \tau_i$, or the symbol ":". This syntactic sugar can be removed by the following source-to-source transformation:

$$t\texttt{!}\,(u_1, \ldots, u_n) \rightarrow \texttt{forall}(x_{i_1}, \ldots, x_{i_k})\texttt{->}\, t\texttt{!}\,(t_1, \ldots, t_n)\ \texttt{at}\ (t'_1, \ldots, t'_n)$$

where u_i: $\texttt{Bnds}\ \tau_i$ or $u_i = \texttt{:}$ precisely when $i \in \{i_1, \ldots, i_k\}$, and:

$$\begin{array}{rcl} u_i\texttt{:}\,\texttt{Bnds}\,\tau_i & \Longrightarrow & t_i = x_i \quad \text{and} \quad t'_i = u_i \\ u_i\texttt{:}\,\tau_i & \Longrightarrow & t_i = u_i \quad \text{and} \quad t'_i = \texttt{all} \\ u_i = \texttt{:} & \Longrightarrow & t_i = x_i \quad \text{and} \quad t'_i = \texttt{all} \end{array}$$

For instance, $\texttt{d!(:,c,m:n)} \rightarrow \texttt{forall(x1,x3)->d!(x1,c,x3) at (all,all,m:n)}$.

A simple example of a definition using these features is matrix multiplication generalised to two-dimensional data fields:

```
dfmult a b = forall(i,j)-> sum a!(i,:)*b!(:,j)
```

Here, `a!(i,:)` is the `i`th row of `a` and `b!(:,j)` the `j`th column of `b`. These one-dimensional data fields are elementwise multiplied and the result is summed. The function defines a two-dimensional data field whose first dimension is constrained by `a`'s bound in the first dimension and by `b`'s bound in the second dimension. This definition works also if `a` and `b` are sparse, although the bound of the result may be an overapproximation since it has to be a product bound. In Sect. 8 we will give larger examples with multidimensional data fields.

7.5 Bounds for Sparse and Dense Arrays

As a concrete example, we now define cpo's of bounds for array-like data fields. We define these cpo's in a way that allows both dense and sparse data fields, and combinations of these: this extends our example well beyond the ordinary dense array model. Since arrays are indexed by flat tuples of integers we will define cpo's of *array bounds* $\mathcal{B}_{arr}(int^n)$ for $n > 0$:

$$\mathcal{B}_{arr}(int) = (int \times int) + Set\ int + Nothing + All + [int \to bool] \quad (14)$$

$$\mathcal{B}_{arr}(int^n) = \mathcal{B}_{arr}(int)^n + [int^n \to bool], \quad n > 1 \quad (15)$$

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. *Set int* is the flat cpo of finite sets of integers. They provide *sparse* one-dimensional array bounds: each element in a set S represents a coordinate where a data field (f, S) is defined. Some examples of bounds in $\mathcal{B}_{arr}(int^2)$ are shown in Fig. 8.

Set int is seen as an abstract data type. We do not specify exactly which elements it contains: rather, we postulate a number of set operations: the usual union ($\cup$), intersection ($\cap$), membership ($\in$), and cardinality ($|\cdot|$), plus removal of element ($\perp$), least element (*least*), elementwise application of function (*smap*), and filtering with predicate (*sfilter*), defined by:

$$\begin{aligned} S \perp i &= \{\, j \mid j \in S \wedge j \neq i \,\} \\ least(S) &= i \text{ where } i \in S \wedge \forall j \in S.j \geq i \quad (S \text{ nonempty}) \\ smap(f, S) &= \{\, f(i) \mid i \in S \,\} \\ sfilter(p, S) &= \{\, i \mid i \in S \wedge p(i) \,\} \end{aligned}$$

These operations are all assumed hyperstrict: for *smap* and *sfilter*, we also assume that the result is $\perp_{Set\ int}$ whenever $f(i) = \perp$ (or $p(i) = \perp$) for some $i \in S$. Finally, we assume that *Set int* contains the empty set $\emptyset$.

Later in this section we will use an abstract data type $Set\ int^n$ for finite subsets of int^n, $n > 0$, with the same abstract operations as on *Set int*. The generalisation is obvious, except that $least(S)$ now should be the least element in S w.r.t. the lexicographical order on int^n.

The finite one-dimensional bounds belong either to $int \times int$, *Set int* or *Nothing*. *all*, and predicates in $[int^n \to bool]$, $n > 0$, are infinite. An n-dimensional bound in $\mathcal{B}_{arr}(int^n)$ is either a product bound or a predicate. We assume that the product bounds have all their properties and operations canonically defined according to Th. 2: this also yields elements *all* and *nothing* for $\mathcal{B}_{arr}(int^n)$.

We have, for predicates $p \in [int^n \to bool]$ ($n > 0$):

$$[\![p]\!]_{bool} = p$$

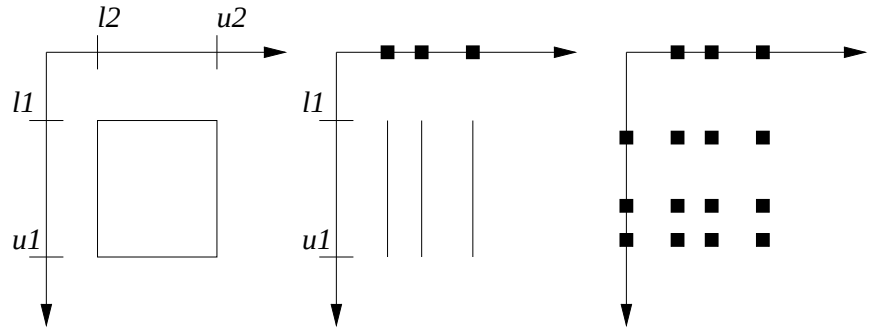

Figure 8: Some bounds in $\mathcal{B}_{arr}(int^2)$, of type: $(int \times int)^2$, $(int \times int) \times Set\ int$, and $(Set\ int) \times (Set\ int)$.

For dense array bounds in $\mathcal{B}_{arr}(int)$ (l, u range over int):

$$\begin{array}{rcl} [\![(l,u)]\!]_{bool} & = & \lambda i.(l \le i \le u) \\ size(l,u) & = & \max(u \perp l + 1, 0) \\ enum(l,u) & = & \lambda i.(i \perp l) \end{array}$$

For sparse array bounds in $\mathcal{B}_{arr}(int)$ (S ranges over $Set\ int$):

$$\begin{array}{rcll} [\![S]\!]_{bool} & = & \lambda i.i \in S & \\ size(S) & = & |S| & \\ enum(S) & = & \lambda i.if\,(i = 0, least(S), enum(S \perp least(S), i \perp 1)) & (S \text{ nonempty}) \end{array}$$

It is next to trivial to verify that the postulated properties for $[\![b]\!]_{bool}$, $size(b)$ and $enum(b)$ holds for all finite nonempty bounds $b \in \mathcal{B}_{arr}(int)$.

We now define $\sqcup$ and $\sqcap$ for bounds in $\mathcal{B}_{arr}(int)$. For all, we specify:

$$\begin{array}{rcll} all \sqcap x & = & x & \\ \perp \sqcap all & = & \perp & \\ b \sqcap all & = & b, & b \in \overline{\mathcal{B}_{arr}(int)} \end{array} \qquad \begin{array}{rcll} all \sqcup x & = & all & \\ \perp \sqcup all & = & \perp & \\ b \sqcup all & = & all, & b \in \overline{\mathcal{B}_{arr}(int)} \end{array} \tag{16}$$

These equations are consistent with a left-to right evaluation order. Note that we do not specify $b \sqcap all$ and $b \sqcup all$ for non-maximal, non-bottom bounds b: this means that we leave the exact strictness properties of $\sqcap$ and $\sqcup$ in their left argument open. We could for instance have $b \sqcup all = b$ for such bounds b (nonstrict evaluation), or $b \sqcup all = \perp$ (hyperstrict evaluation).

We define $\sqcup$ and $\sqcap$ for *nothing* as the exact dual of (16): replace *all* with *nothing* and switch $\sqcup$ and $\sqcap$.

For one-dimensional dense array bounds, we define

$$\begin{array}{rcl} (l,u) \sqcap (l',u') & = & (\max(l,l'), \min(u,u')) \\ (l,u) \sqcup (l',u') & = & (\min(l,l'), \max(u,u')) \end{array}$$

If $b, b' \in (int \times int) + Set\ int$ and at least one of them belongs to $Set\ int$, then

$$\begin{array}{rcl} b \sqcap b' & = & toset(b) \cap toset(b') \\ b \sqcup b' & = & toset(b) \cup toset(b') \end{array}$$

where $toset$: $[(int \times int) + Set\ int \to Set\ int]$ is defined by

$$\begin{array}{rcl} toset(b) & = & toset'(b,0), \quad \text{where} \\ toset'(b,n) & = & if\,(n = size(b), \emptyset, \{enum(b,n)\} \cup toset'(b, n+1)) \end{array}$$

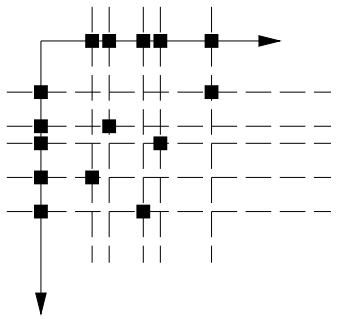

Figure 9: Finite set $S \in Set\, int^2$, and projections $\pi_1(S)$, $\pi_2(S)$.

If $b \in (int \times int) + Set\, int$ and $b' \in [int \to bool]$, then

$$\begin{array}{lcccl} b \sqcap b' & = & b' \sqcap b & = & sfilter([\![b']\!]_{bool}, toset(b)) \\ b \sqcup b' & = & [\![b]\!]_{bool} \vee_X b' & & \\ b' \sqcup b & = & b' \vee_X [\![b]\!]_{bool}. & & \end{array}$$

If $b, b' \in [int \to bool]$, then

$$\begin{array}{lcl} b \sqcap b' & = & b \wedge_X b' \\ b \sqcup b' & = & b \vee_X b'. \end{array}$$

The remaining cases for $\sqcup$ and $\sqcap$ for bounds in $\mathcal{B}_{arr}(int^n)$, $n > 1$, are defined in the following way. If $b \in [int^n \to bool]$ and $b' = b'_1 \times \cdots \times b'_n$, then

$$\begin{array}{lcl} b \sqcup b' & = & b \vee_X [\![b']\!]_{bool} \\ b' \sqcup b & = & [\![b']\!]_{bool} \vee_X b \end{array}$$

If b' is finite, then

$$b \sqcap b' = b' \sqcap b = \times_{i=1}^{n} \pi_i(sfilter(b, toset(b')))$$

where the projections $\pi_i \in [Set\, int^n \to Set\, int]$, $1 \leq i \leq n$, are defined by $\pi_i(S) = \{\, j \mid \exists \vec{x} \in S. x_i = j \,\}$ and $toset(b')$ is generalized into a function in $[((int \times int) + Set\, int + Nothing)^n \to Set\, int^n]$ in the obvious way (its definition carries over verbatim). See Fig. 9. If b' is infinite, then

$$\begin{array}{lcl} b \sqcap b' & = & b \wedge_X [\![b']\!]_{bool} \\ b' \sqcap b & = & [\![b']\!]_{bool} \wedge_X b \end{array}$$

If, finally, $b, b' \in [int^n \to bool]$, then

$$\begin{array}{lcl} b \sqcap b' & = & b \wedge_X b' \\ b \sqcup b' & = & b \vee_X b'. \end{array}$$

The definitions above should be seen as abstract semantical definitions. Implementations should of course use more efficient, specialized versions of functions and representations whenever possible. For instance, it is easy to verify that $toset(S) = S$ for finite sets S.

Finally, let us verify that $\sqcap$, $\sqcup$ as defined above do enjoy the postulated properties in Definition 4:

Proposition 13 *For all maximal b, b' in $\mathcal{B}_{arr}(int^n)$ it holds that $b \sqcap b'$ and $b \sqcup b'$ both are maximal, $\{\{b\}\} \cap \{\{b'\}\} \subseteq \{\{b \sqcap b'\}\}$, and $\{\{b\}\} \cup \{\{b'\}\} \subseteq \{\{b \sqcup b'\}\}$.*

Proof. It is trivial that $b \sqcap b'$ and $b \sqcup b'$ are maximal whenever b and b' are. Similarly, for most cases it is trivial to prove that $\{\{b\}\} \cap \{\{b'\}\} \subseteq \{\{b \sqcap b'\}\}$ and $\{\{b\}\} \cup \{\{b'\}\} \subseteq \{\{b \sqcup b'\}\}$ for maximal b, b'. The possibly nontrivial case is when $b \sqcap b' = b' \sqcap b = \times_{i=1}^{n} \pi_i(\mathit{sfilter}(b, \mathit{toset}(b')))$. But then, we can first note that $\{\{b\}\} \cap \{\{b'\}\} = \mathit{sfilter}(b, \mathit{toset}(b'))$ whenever b and b' are maximal, and then that $S \subseteq \times_{i=1}^{n} \pi_i(S)$ for all $S \in \mathit{Set}\, \mathit{int}^n$. This yields the result also in this case. ■

7.6 Sparse Multidimensional Bounds and Relational Databases

So far, we have considered mainly multidimensional product bounds. We now turn to finite multidimensional sparse bounds and how operations on them could be defined. At first sight, it may seem that we simply could define sparse n-dimensional bounds as members of the abstract data type $\mathit{Set}\,(\alpha_1 \times \cdots \times \alpha_n)$ and define the operations on bounds in terms of the abstract set operations on this data type. However, this data type cannot be closed under the operations we consider. To see this, reconsider the expression $t = \varphi(x_1, x_2, x_3).(f, b)\,!\,(x_2, c, x_1, x_1)$. In Sect. 7.3 we considered the case where b is a product bound and arrived at a bound for t where x_3 is unconstrained. This should still hold if b is a sparse bound. x_2 and x_3, on the other hand, should be constrained by the following: the bound for t should contain only tuples (x_1, x_2, x_3) where there exists an element $(e_1, e_2, e_3, e_4) \in \{\{b\}\}$ such that $e_1 = x_2$, $e_2 = c$, and $e_3 = e_4 = x_1$. This is an infinite set, but since b is finite it can be given a finite representation as a sparse set of two-dimensional tuples (x_1, x_2), obeying the above, which defines a three-dimensional bound which is unconstrained in the third dimension.

We thus need bounds which are sparse finite sets embedded into a higher-dimensional space. To define these it is convenient to generalise n-tuples into records indexed by attribute sets: $x = (\, x.a \mid a \in A \,)$ where A is a finite set of attribute names. (n-tuples can then be seen as records with attribute set $\{1, \ldots, n\}$.) Semantically, these records are functions in $[A \to \bigcup_{a \in A} D_a]$, where $x.a \in D_a$ for $a \in A$. Sparse sets of records are elements in $\mathit{Set}\,[A \to \bigcup_{a \in A} D_a]$. For such sets S we write $\mathit{Attr}(S)$ for A. In light of the above, we define that if $S \in \mathit{Set}\,[A \to \bigcup_{a \in A} D_a]$ then $S \in \mathcal{B}([A' \to \bigcup_{a \in A'} D'_a])$ whenever $A \subseteq A'$ and $D'_a = D_a$ for $a \in A$. We now define, somewhat informally, the operations in Definition 4 on bounds in $\mathcal{B}([A' \to \bigcup_{a \in A'} D'_a])$ as follows (cf. Sect. 7.5):

- $\mathit{finite}(S)$ iff $\mathit{Attr}(S) = A'$
- $[\![S]\!]_{\mathit{bool}} = \lambda x.\exists y \in S.\forall a \in \mathit{Attr}(S).\, x.a = y.a$
- Enumeration according to lexical order of records (we assume a total order on A)
- $\mathit{size}(S) = |S|$ when $\mathit{Attr}(S) = A'$

In order to define $\sqcup$, $\sqcap$, and $\mathit{bproj}_{p,I}$ we introduce the following operations, known from relational database theory (see, for instance, [49]):

$$\begin{aligned} \mathit{Project}(S, A) &= \{\, x \mid \exists y \in S.\forall a \in A.\, x.a = y.a \,\} \\ \mathit{Join}(S_1, S_2) &= \{\, t \mid (\exists t_1 \in S_1.\forall a \in \mathit{Attr}(S_1).\, t.a = t_1.a) \wedge \\ &\qquad (\exists t_2 \in S_2.\forall a \in \mathit{Attr}(S_2).\, t.a = t_2.a) \,\}. \end{aligned}$$

We can then define:

- $S_1 \sqcap S_2 = \mathit{Join}(S_1, S_2)$

- $S_1 \sqcup S_2 = Project(S_1, Attr(S_1) \setminus Attr(S_2)) \cup Project(S_2, Attr(S_2) \setminus Attr(S_1))$
- $bproj_{p,I}(S,t) = \{\, (x.b \mid b \in rg(p)\,) \mid \exists y \in S. \forall a \in dom(p).\, x.p(a) = y.a \wedge \forall a \in Attr(S) \setminus dom(p) \setminus I.\, x.p(a) = t.a \,\}$

Here, $I \subseteq Attr(S)$. $rg(p)$ stands for the range of p. We omit the verification that these operations do have the the properties required in Definition 4 and in (12): it is fairly straightforward but tedious to carry out.

Relational databases can be seen as sparse sets of records indexed by attributes. Thus, it is possible to define data fields whose bounds are databases. Let us consider briefly how our data field language from Sect. 3.1, extended with records and bounds which are sparse sets of records, can be used as a language for querying and computing over relational databases. We introduce an obvious syntax for records:

$$(a_1 : t_1\,, \ldots\,, a_n : t_n)$$

Forall-syntax with pattern-matching on records is straightforward to define, as well as syntactic sugar for selection/projection similar to the sugaring for data fields with product bounds defined in Sect. 7.4.

In order to use data field primitives to compute with databases we need a function to "lift" a database into a data field:

```
df b = (\x->x,b)
```

To create a data field of all the values of attribute `a` from database `s`, we write

```
forall x-> ((df s)!x).a
```

or, using elemental intrinsics overloading, simply

```
(df s).a
```

We can now, for instance, write `hist (df s).a` for the histogram over the values of the attribute.

Now consider a database `s` with attribute set {`ssn`, `salary`, `age`}, and a data field `frac_of_inc` over a database with attribute set {`ssn`, `expense`}. The setting could be that `s` is a database over individuals, each identified by its `ssn` and having possibly several incomes, and that `frac_of_inc` tabulates, for each individual, the fractions of the total income spent on different kinds of expenses. Maybe we would like to compute, for all individuals of age 43 who have any income, how they spend their money in absolute figures. The following `forall`-expression defines a data field with this information:

```
forall(id:x1, exp:x2)-> (sum (df s)!(ssn:x1, age:43).salary) *
                          frac_of_inc!(ssn:x1, expense:x2)
```

How does this work? The term being summed over has the attribute `salary` omitted. As for the syntax in Sect. 7.4 we transform this term into

```
forall x-> (df s)!(ssn:x1, salary:x, age:43).salary
```

Informally, the net effect is that for each `x1` the sum of all salaries is computed, over each set of records `r` with `r.ssn` = `x1` and `r.age` = 43. As the second factor we select, for the same `x1` and each possible `x2`, the corresponding entry in `frac_of_inc`. The result is a data field over a database with attributes `id` and `exp` which holds a table over each individual of age 43 and his income split on different expenses.

Formally, the following happens. The datafield being summed over has, for any given value x_1 of `x1`, the bound $bproj_{p,\emptyset}$(`s`, (`ssn:`x_1`, salary:x, age:43`))

with $dom(p) = \{\texttt{salary}\}$, and $p(\texttt{salary}) = 1$. This bound equals $\{\, z \mid \exists y \in \texttt{s}.\, y.\texttt{salary} = z \wedge y.\texttt{ssn} = x_1 \wedge y.\texttt{age} = 43 \,\}$. For the first factor, with respect to the outermost `forall`, the bound $bproj_{p',\{\texttt{x}\}}(\texttt{s}, \texttt{(ssn:x1, salary:x, age:43)})$ is obtained, with $dom(p') = \{\texttt{ssn}\}$, and $p'(\texttt{ssn}) = \texttt{id}$. This bound is the set $S_1 = \{\, (z.\texttt{id}) \mid \exists y \in \texttt{s}.\, y.\texttt{ssn} = z.\texttt{id} \wedge y.\texttt{age} = 43 \,\}$. Note that S_1 does not constrain `x2` and is thus infinite. The second factor, finally, has the bound $bproj_{p'',\emptyset}(S, \texttt{(ssn:x1, expense:x2)})$, where S is the bound of `frac_of_inc`, with $dom(p'') = \{\texttt{ssn}, \texttt{expense}\}$, $p''(\texttt{ssn}) = \texttt{id}$, and $p''(\texttt{expense}) = \texttt{exp}$. This bound equals $S_2 = \{\, (z.\texttt{id}, z.\texttt{exp}) \mid \exists y \in S.\, z.\texttt{id} = y.\texttt{ssn} \wedge z.\texttt{exp} = y.\texttt{expense} \,\}$. The bound for the whole `forall`-expression is $S_1 \sqcap S_2 = Join(S_1, S_2)$, which is a finite set constraining both `x1` and `x2`. Thus, it is a finite bound even though S_1 is infinite.

7.7 Translations and Scalings of Sparse/Dense Arrays

Many array operations require that arrays, with their bounds, are translated w.r.t. some constant offset vector. Other operations require that arrays are reversed, or accessed with some constant stride. These operations can be seen as a scaling of the array, possibly with a negative factor. B_0 and B_1 do not define implicit propagation of bounds w.r.t. these operations. We now propose an extension B_2 which defines this for our sparse/dense array bounds from Sect. 7.5. (This could also easily be done for the sparse multidimensional bounds in Sect. 7.6.)

Translation and scaling of a data field d can be expressed as $\varphi\vec{x}.d!g(\vec{x})$, where g is an affine function. So we should define $B_2((f,b)\,!\,g(\vec{x}), \vec{x}, Y)$ when g is affine. What should it be? For bounds which define finite sets, we can state some general facts. Consider a general function g, bound b, and function G taking bounds to bounds such that $B_2((f,b)\,!\,g(\vec{x}), \vec{x}, Y) = G(b)$. We want to have $\{\{[\![b]\!]_{bool} \circ g\}\} \subseteq \{\{G(b)\}\}$: then we can use Lemma 1 to extend Lemma 2 to cover B_2, which in turn extends Th. 1 to B_2. We have $\{\{[\![b]\!]_{bool} \circ g\}\} = \{\, x \mid g(x) \in \{\{b\}\} \,\}$. If g is invertible, then this set equals $\{\, g^{-1}(x) \mid x \in \{\{b\}\} \cap Im(g) \,\} = b'$ (here, $Im(g)$ stands for the image of g). Now, if b is a finite bound and we can compute g^{-1}, then we can compute $G(b)$ from b in the following way:

- directly as b', if $G(b)$ is a sparse bound,
- as $(\min(\, x \mid x \in b' \,), \max(\, x \mid x \in b' \,))$, if $G(b)$ is a dense bound.

In the following, we will use elemental intrinsics overloading of addition and multiplication on integer tuples, that is: $\vec{t} + \vec{t'} = (t_1 + t'_1, \ldots, t_n + t'_n)$ and similarly for $\vec{t} \cdot \vec{t'}$. We also define $\vec{n} = (n, \ldots, n)$ for numerical constants n. First, we define translations of bounds:

$$B_2((f,b)\,!\,(\vec{x} + \vec{a}), \vec{x}, Y) = tr(b, \vec{a}), \quad FV(\vec{a}) = \emptyset \tag{17}$$

$tr(b, \vec{a})$ is defined as follows, for the different forms of b:

$$\begin{aligned}
tr((l,u),a) &= (l \perp a, u \perp a), \quad (l,u) \in int \times int && (18)\\
tr(S,a) &= smap(\lambda x.(x \perp a), S), \quad S \in Set\ int && (19)\\
tr((b_1,\ldots,b_n),(a_1,\ldots,a_n)) &= (tr(b_1,a_1),\ldots,tr(b_n,a_n)) && (20)\\
tr(all,\vec{a}) &= all && (21)\\
tr(nothing,\vec{a}) &= nothing && (22)\\
tr(p,\vec{a}) &= \lambda\vec{x}.p(\vec{x}+\vec{a}), \quad p \in int^n \to bool && (23)
\end{aligned}$$

See Fig. 10 for a simple example. Also note that $tr(b, \vec{0}) = b$.

The correctness of tr is stated in the following proposition, which is straightforward to prove.

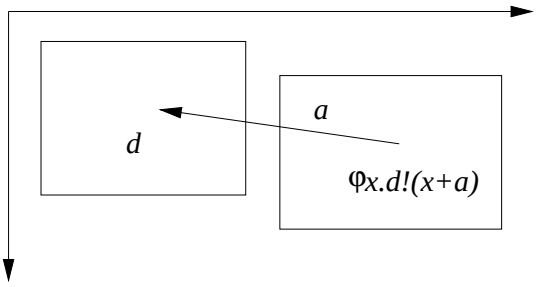

Figure 10: Translation of data field d to $\varphi x.d\,!\,(x+a)$.

Proposition 14 $\{\{[\![b]\!]_{bool} \circ (\lambda\vec{x}.(\vec{x}+\vec{a}))\}\} = \{\{tr(b,\vec{a})\}\}$.

Next, we define scaling:

$$B_2((f,b)\,!\,(\vec{z}\cdot\vec{x}),\vec{x},Y) = sc(\vec{z},b), \quad FV(\vec{z}) = \emptyset \tag{24}$$

$sc(\vec{z},b)$ is defined as follows, for the different forms of b:

$$\begin{aligned}
sc(0,b) &= if([\![b]\!]_{bool}(0), all, nothing), \quad \text{all forms of } b && (25)\\
sc(z,(l,u)) &= (\lceil l/z\rceil, \lfloor u/z\rfloor), \quad (l,u)\in int\times int, z>0 && (26)\\
sc(z,(l,u)) &= (\lceil u/z\rceil, \lfloor l/z\rfloor), \quad (l,u)\in int\times int, z<0 && (27)\\
sc(z,S) &= smap(\lambda x.(x/z), sfilter(\lambda x.(x/z=\lfloor x/z\rfloor), S)),\\
&\quad S\in Set\ int, z\neq 0 && (28)\\
sc((z_1,\ldots,z_n),(b_1,\ldots,b_n)) &= (sc(z_1,b_1),\ldots,sc(z_n,b_n)) && (29)\\
sc(\vec{z},all) &= all && (30)\\
sc(\vec{z},nothing) &= nothing && (31)\\
sc(\vec{z},p) &= \lambda\vec{x}.p(\vec{z}\cdot\vec{x}), \quad p\in int^n\to bool, z\neq 0 && (32)
\end{aligned}$$

Note that (25) simplifies into (30) and (31) for *all* and *nothing*, respectively. Also note that $sc(\vec{1},b) = b$.

As for tr, there is a correctness result for sc. It is tedious but straightforward to prove, so we omit the proof:

Proposition 15 $\{\{[\![b]\!]_{bool} \circ (\lambda\vec{x}.\vec{z}\cdot\vec{x})\}\} = \{\{sc(\vec{z},b)\}\}$.

Finally, we combine scaling, translation, and the "selection/projection" of Section 7.2 into a more general case for $B_2(t,\vec{x},Y)$. In order to do this, we need the following result which is easy to prove.

Proposition 16 *If* $\{\{[\![b]\!]_{bool}\circ g\}\}\subseteq\{\{G(b)\}\}$ *and* $\{\{[\![b]\!]_{bool}\circ h\}\}\subseteq\{\{H(b)\}\}$ *for all bounds* b*, then* $\{\{[\![b]\!]_{bool}\circ g\circ h\}\}\subseteq\{\{H(G(b))\}\}$ *for all bounds* b.

We now make the following definition.

$$B_2((f,b)\,!\,(\vec{z}\cdot\vec{t}[p,\vec{x}]+\vec{a}),\vec{x},Y) = tr(sc(\vec{z},bproj_{p,I}(b,\vec{t})),\vec{a}), \tag{33}$$

where $\emptyset\subset FV(t_i)\subseteq Y$ for $i\in I$, $FV(t_i)=\emptyset$ for $i\notin I$, and $FV(\vec{z}) = FV(\vec{a}) = \emptyset$. Finally, we extend (10.1) into (10.2):

$$B_2((f,b)\,!\,\vec{t},\vec{x},Y) = B_2(\vec{t},\vec{x},Y), \quad FV(f,b)=\emptyset, \vec{t}\neq\vec{z}\cdot\vec{t}[p,\vec{x}]+\vec{a} \tag{10.2}$$

for all $\vec{z}\cdot\vec{t}[p,\vec{x}]+\vec{a}$ as defined in (33). If we consider $\vec{t}[p,\vec{x}]$ as equal to $\vec{1}\cdot\vec{t}[p,\vec{x}]$ and $\vec{z}\cdot\vec{t}[p,\vec{x}]$ as equal to $\vec{z}\cdot\vec{t}[p,\vec{x}]+\vec{0}$, then (33) and (10.2) subsume (13), (24), (17), and (10.1).

Definition 12 *$B_2(t, \vec{x}, Y)$ is defined by equations (2.1) – (8.1), (10.2), (11.1), and (33).*

As for B_1 Lemma 2 and thus Th. 1 can be extended, through Propositions 14, 15, and 16, to cover B_2. We omit the details.

7.8 An Example: Data Fields for Symbolic Drawings

We now describe how data fields could be used to define drawings comprised of symbolic objects. Some operations on these objects will rely on the transformation of bounds under translation and scaling in Sect. 7.7. We will consider two kinds of data fields for this purpose:

- *scenes*, which are functions describing images in real scale. These are represented as data fields of type `Df (Float,Float) a`;
- *bitmaps*, which are "sampled" scenes represented as data fields of type `Df (Int,Int) a`.

`a` is some type which describes the image property in each point (e.g., colour, intensity). Scenes are built from objects, which also are data fields of type `Df (Float,Float) a`. In this context, the data field defines an image property for each point in the plane where it is defined, and can be seen as "transparent" in points where it is undefined. The *finite*, *enum*, and *size* functions on bounds are not needed for objects and scenes since these will be evaluated only when "viewed" through a bitmap. We can imagine a superclass to data fields for which these functions need not be defined. It is even possible to simply use functions of type `(Float,Float) -> a` to represent objects and scenes, but for efficiency reasons (like if objects are moved w.r.t. a mutable bitmap, see Sect. 7.10) it may make sense to have bounds for objects which can act as "bounding boxes".

The following function computes a `m` × `n`-bitmap from a scene, with a scale factor for pixels per length unit and an offset which defines the origin of the bitmap:

```
bitmap scene scale (off_i,off_j) col m n =
  forall (i,j)-> (bg col scene!(i/scale+off_i,j/scale+off_j))
                 at (1:m,1:n)

bg col x = if isoub x then col else x
```

See Fig. 11. The elementwise applied `bg col` defines a background colour. (We use Haskell style pattern-matching on arguments in the definition of `bitmap`.)

How are objects composed into scenes? We define a connective `over` which puts an object above another:

```
d1 `over` d2 = forall x-> if not (isoub d1!x) then d1!x else d2!x
```

Objects can be transformed in different ways. For instance, the function below changes the colour of an object:

```
dye d col = (forall x-> if not (isoub d1!x) then col else oub)
            at bounds d
```

Note the explicit restriction: the data field defined by the `forall`-expression has bound `all`. The function `bounds` is defined by:

```
bounds (_,b) = b
```

Another group is geometrical transformations, e.g., translating an object:

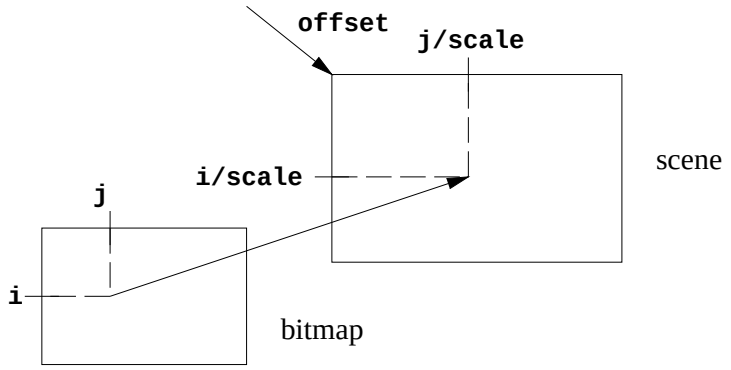

Figure 11: Connecting bitmaps with scenes.

```
translate d (off_x,off_y) = forall (x,y)-> d!(x-off_x,y-off_y)
```

Other geometrical translations require that objects have some origin. If we consider more complex objects which are pairs of objects and origins (pairs of floats), then we may for instance define a function which "flips" objects in the first dimension:

```
flip_x (d,(ox,oy)) = (forall (x,y)-> d!(ox-x,y),(ox,oy))
```

It is straightforward to redefine the previous operations on objects and scenes which are simple data fields so they work also on objects with origins.

7.9 Alternative φ-calculi for Conformance

Most existing array languages require that the operands of elementwise applied operations are conformant, that is, that their extents are the same after they have been aligned. The model developed here is different – it yields the implicit intersection rule for elementwise applied strict operations, and a related rule for elementwise applied conditional. It is, however, simple to modify the φ-calculi given here to yield conformance instead. In order to obtain conformant versions of the calculi which are independent of the type of bounds, the following is needed:

- A new required operation on bounds in Definition 4, which is a test "=" for equality of bounds, and
- The following rewrite rule, which replaces (5) and (6):

$$B_i(op(t_1,\ldots,t_m),x,Y) = if(\bigwedge_{j=1}^{m}\bigwedge_{k=1}^{m} B_i(t_j,x,Y) = B_i(t_k,x,Y), B_i(t_1,x,Y), *)$$

These modifications alone do not yield the alignment of array operands that typically takes place in array languages before the extents of the operands are matched. This alignment is strongly tied to the dense array type and makes little sense for other indexed data structures. Actually, to avoid ambiguities about which operand to align with which, one should probably define this alignment only for array types where the lower bound is fixed (say, 0). The alignment then becomes a matter of defining the array operations in the language so they always yield arrays of this kind. We leave the details for the interested reader to work out.

7.10 Mutable Data Fields

Efficiency is often a concern. Many languages therefore provide mutable arrays which can be updated in place. We now outline how mutable data fields could be defined.

For simplicity we do this in a simple imperative language (similar to IMP in [70]) extended to allow concurrent assignments of data fields, with types and terms according to Sect. 3.1. We only give enough details to make the point. The language has typed program variables, and states which are mappings from program variables to values. In particular, program variables of type `Df` $\tau_1\ \tau_2$ hold data fields. For any data field d we use the notation d_f and d_b for its function and bound, respectively (i.e., $d = (d_f, d_b)$).

The meaning of a program c is a function $\mathcal{C}[\![c]\!]$ which maps states to states, and for any term there is a function which maps states to values of the correct type. In particular $\mathcal{F}[\![t]\!]$ maps data field-typed terms to data fields. For states σ, $\sigma[v/x]$ is defined by $\sigma[v/x](x) = v$, and $\sigma[v/x](y) = \sigma(y)$ whenever $y \neq x$.

The language has assignments x `:=` t, where x is a program variable and t is a term. Assignments of data field-typed variables can be seen as a concurrent assignment of some or all of their elements. To make in-place update possible it is important that *these variables don't have their bounds changed*. To accomplish this, we can give semantics to data field assignments in one of the two following ways:

- Requiring conformance: $\mathcal{C}[\![x := t]\!]\sigma = \mathit{if}\,((\mathcal{F}[\![t]\!]\sigma)_b = \sigma(x)_b, \sigma[\mathcal{F}[\![t]\!]\sigma/x], *)$
- Updating only the elements where the right-hand side is defined: $\mathcal{C}[\![x := t]\!]\sigma = \sigma[(\lambda y.\mathit{if}\,((\![\mathcal{F}[\![t]\!]\sigma]\!])_b, (\mathcal{F}[\![t]\!]\sigma)_f\ y, \sigma(x)_f\ y), \sigma(x)_b)/x]$

(Cf. denotational semantics for "ordinary" assignments [70].) Defining the semantics in this way has the advantage that the issue of mutability becomes largely orthogonal to the exact semantics of data field expressions.

8 Examples

We now exemplify the use of data fields for the specification of parallel algorithms. We will use the simple functional data field language developed in Sections 3.1 and 7.4, extended with some conveniences. We give three examples: Strassen's algorithm for matrix multiplication, which is a recursive divide-and-conquer style block-structured matrix algorithm, data parallel LU factorization with pivoting, which is an array algorithm with data dependent structure, and a sparse parallel neural network algorithm. An early version of the latter algorithm was presented in [23].

8.1 Extensions of the Data Field Language

First we add ordinary `if..then..elseif..else` as shorthand for nested conditionals. Then, we define a notation to define different parts of data fields by cases. The expression

$$\texttt{case}\,(b_1\ \texttt{->}\ t_1\,;\ \ldots\ ;\ b_n\ \texttt{->}\ t_n\,;\ \texttt{otherwise}\ t)$$

is syntactic sugar for

$$\begin{array}{ll}\texttt{forall}\ x\texttt{->} & \texttt{if in}\ b_1\ x\ \texttt{then}\ (t_1\ \texttt{at}\ b_1)\,!\,x\\ & \texttt{elseif in}\ b_2\ x\ \texttt{then}\ (t_2\ \texttt{at}\ b_2)\,!\,x\\ & \vdots\\ & \texttt{elseif in}\ b_n\ x\ \texttt{then}\ (t_n\ \texttt{at}\ b_n)\,!\,x\\ & \texttt{else}\ t\end{array}$$

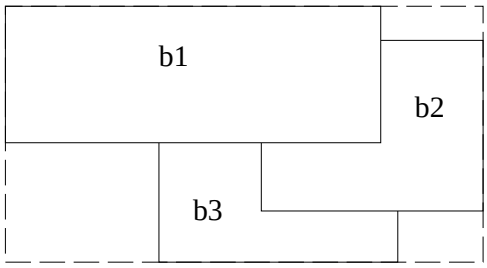

Figure 12: Illustration of `case` $(b_1$ `->` t_1 ; b_2 `->` t_2 ; b_3 `->` $t_3)$.

(We may allow an empty `otherwise` which is equivalent to t = `forall x->oub`.) See Fig. 12 for an illustration.

Furthermore, we define notation for 2-dimensional data fields `d`: `Df (Int,Int)` τ with canonical product bounds. (It is easily extended to n-dimensional data fields.) When these bounds are finite and nonempty we define, for $i = 1, 2$:

```
l_i (_,b) = enum (proj_i b) 0
u_i (_,b) = enum (proj_i b) (size b - 1)
align d = forall(x1,x2)-> d!(x1-(l_1 d)-1,x2-(l_2 d)-1)
++_1 d1 d2 = forall(x1,x2)->
   if x1 > u_1 d1 then d2!(x1-(u_1 d1)+(l_1 d2)-1,x2) else d1!(x!,x2)
++_2 d1 d2 = forall(x1,x2)->
   if x2 > u_2 d1 then d2!(x1,x2-(u_2 d1)+(l_2 d2)-1) else d1!(x!,x2)
```

`l_`i and `u_`i give lower and upper limit, respectively, in direction i. `align d` returns `d` with the left-hand upper corner aligned with $(1, 1)$. `++_`i is data field concatenation in direction i. Finally, for convenience, we define:

```
first b = enum b 0
```

We will also make use of some other conveniences like `let`-constructs.

8.2 Strassen's Matrix Multiplication

Strassen's matrix multiplication [63] is famous since it was the first known matrix multiplication algorithm with complexity strictly less than $O(n^3)$ for $n \times n$-matrices. It is a recursive block-structured algorithm where the matrices are successively split in four similar blocks. When the matrices are of size $n \times n$, where $n = 2^m$ for some natural number m, the blocks will always have the same size. We restrict our presentation to this case.

Consider the matrix product $C = AB$. In a block formulation

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

there are four computations like

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$

to perform, which yields 8 matrix multiplies and 4 matrix adds. Using Strassen's method we can perform the same computation using 7 matrix multiplies and 18 matrix adds. The following operations are performed, where the method is applied

recursively for the multiplications of blocks:

$$
\begin{array}{lcllcl}
m_1 & = & (a_{12} \perp a_{22})(b_{21} + b_{22}) & c_{11} & = & m_1 + m_2 \perp m_4 + m_6 \\
m_2 & = & (a_{11} + a_{22})(b_{11} + b_{22}) & c_{12} & = & m_4 + m_5 \\
m_3 & = & (a_{11} \perp a_{21})(b_{11} + b_{12}) & c_{21} & = & m_6 + m_7 \\
m_4 & = & (a_{11} + a_{12})b_{22} & c_{22} & = & m_2 \perp m_3 + m_5 \perp m_7 \\
m_5 & = & a_{11}(b_{12} \perp b_{22}) & & & \\
m_6 & = & a_{22}(b_{21} \perp b_{11}) & & & \\
m_7 & = & (a_{21} + a_{22})b_{11} & & &
\end{array}
$$

We now define a data field function for Strassen's algoritm, where matrices are represented by data fields restricted by bounds in $(int \times int)^2 \subseteq \mathcal{B}_{arr}(int^2)$. Strassen's algorithm operates on square matrices with upper left corner $(1,1)$: thus, we will make heavy use of the alignment function. It is also convenient to define functions which select the four different (north, south) $\times$ (east, west) subfields of a $2^m \times 2^m$-data field:

```
let
  n d = l_1(d):u_1(d)/2
  s d = u_1(d)/2+1:u_1(d)
  e d = l_2(d):u_2(d)/2
  w d = u_2(d)/2+1:u_2(d)
in
ne d = align (d at (n d,e d))
nw d = align (d at (n d,w d))
se d = align (d at (s d,e d))
sw d = align (d at (s d,w d))
```

With these definitions, the data field function is defined viz.

```
strassen a b =
  if size (bounds a) == 1 then a*b
  else
  let
    m1 = strassen ((nw a)-(sw a)) ((se b)+(sw b))
    m2 = strassen ((ne a)+(sw a)) ((ne b)+(sw b))
    m3 = strassen ((ne a)-(se a)) ((ne b)+(nw b))
    m4 = strassen ((ne a)+(nw a)) (sw b)
    m5 = strassen (ne a) ((nw b)-(sw b))
    m6 = strassen (sw a) ((se b)-(ne b))
    m7 = strassen ((se a)+(sw a)) (ne b)
    c11 = m1 + m2 - m4 + m6
    c12 = m4 + m5
    c21 = m6 + m7
    c22 = m2 - m3 + m5 - m7
  in
    (c11 `++_1` c21) `++_2` (c12 `++_1` c22)
```

Note that the resulting code is void of any explicit bounds.

8.3 Data parallel LU factorisation

LU factorisation is a classical problem in linear algebra [13]. The task is to factorise an $n \times n$-matrix $A = LU$ where L is a lower-triangular matrix and U is upper-triangular. The standard algorithm computes L and U in such a way that the diagonal elements of L all are equal to one, and stores U and the nondiagonal

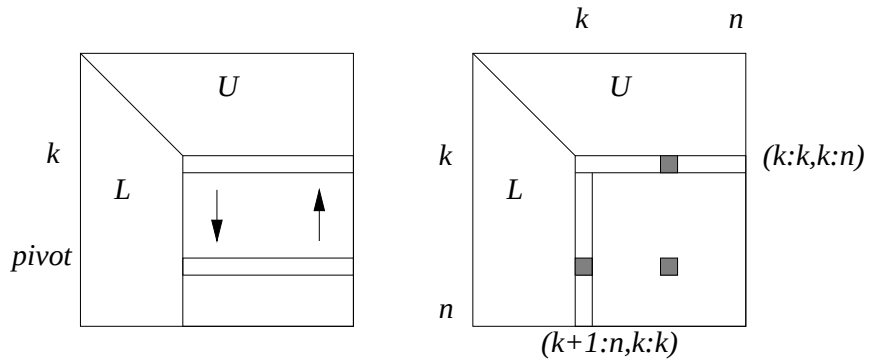

Figure 13: Pivoting and recursive assembly of result in the LU factorisation algorithm.

elements of L in a single $n \times n$-matrix. This algorithm has quite some inherent parallelism, which can be revealed by a collection-oriented programming style with extensive use of elementwise applied operations. Array languages like HPF can express this part well. However, the pivoting (data-dependent exchange of rows as to minimise the numerical error) is not so straightforward to express in a collection-oriented way in these languages. Data fields with sparse and dense array bounds can be used for this purpose.

The following is an informal description of the classical algorithm. The input is the $n \times n$-matrix A_1. For $k = 1$ to $n \perp 1$:

1. Shift rows in A_k as to bring to the top the row whose first element has the greatest magnitude. (Pivoting)
2. If this element (the *pivot element*) is zero, return with error.
3. Divide first column (except first element) with pivot element. Negate.
4. For all elements in $A_k(k+1:n, k+1:n)$, add corresponding element in first row times corresponding element in first column.
5. Apply recursively to $A_{k+1} = A_k(k+1:n, k+1:n)$, concatenate results.

The data field formulation follows. For this algorithm it is natural to let results "stay in place" rather than aligning their upper left corners. Therefore, the assembly of the result can be done by a simple `case` rather than data field concatenation. We define a function for exchanging two rows in a matrix:

```
swap a k l = forall(i,j)-> if i==k
                           then (a!(:,l:l))!(i-k+l,j)
                           elseif i=l then (a!(:,k:k))!(i-l+k,j)
                           else a
```

Below is the data field function for LU factorisation.

```
lu a = if size (bounds a) == 1 then a
       else
         let
           k = l_1 a
           n = u_1 a
           pivot = first (bounds a!(:,k)) at
```

```
          abs a!(:,k) == fold max (abs a!(:,k)) minnum
in
  if a!(pivot,k) == 0.0 then *
  else
    let
      a' = swap a k pivot
    in
      case((k:k,k:n) -> a';
           (k+1:n,k:k) -> -a'/a'!(k,k);
           otherwise ->
             LU (a'+forall(i,j)->
                    (a'!(k,k+1:n))!i * a'!(k+1:n,k)!j)
          )
```

See Fig. 13 for an illustration of some steps in the algorithm. Also note how the location of the pivot element is found by first generating a sparse data field defined exactly in the (possibly several) locations where the element of `a!(k,:)` has maximal magnitude, and then picking the first of these locations (`minnum` is a constant representing the least number for a numeric data type of bounded size). This is a fairly generic way to find the location of a data field element satisfying a certain condition.

8.4 A Sparse Neural Network

Neural networks can be sparse both in connections and activity. Sparse connectivity means that not all neurons are connected and sparse activity means that not all neurons and synapses are active all the time.

Artificial neural networks in the SANS model [41] can, when stimulated with an input, recall a stored pattern by an iterative relaxation method. Let I be the set of units in the network. The patterns are stored as biases β_i and weights w_{ij}, for $i, j \in I$. An input can be fed into the network which is then iterated until a termination criterion is fulfilled. The following equations are used as a basis for the iteration:

$$s_i = \beta_i + \sum_j w_{ij}\pi_j, \tag{34}$$

$$\frac{dE_i}{dt} = \frac{s_i \perp E_i}{\tau_E}, \tag{35}$$

$$\pi_i = f(E_i, \beta_i) = \begin{cases} 0 & E_i \leq \beta_i \\ e^{E_i} & \beta_i < E_i \leq 0 \\ 1 & 0 < E_i \end{cases} \tag{36}$$

E_i is a slowly changing "activity" of unit i, π_i is the unit's output, and τ_E is the time constant of the units.

In the iteration, new activities and outputs are calculated for the units from the old ones. Typically, after an initial phase, most elements will have E_i as either 0 or 1 and very few will change, which means that very few outputs π_i will change. A parallel implementation will therefore benefit from communicating *differences* between new and old outputs rather than the absolute output levels: most differences will be zero and need therefore not be sent, which greatly reduces the communication and also the arithmetics needed when updating the activities. See [22]. The following is an iterative data field algorithm:

```
recall beta w tau_e pi_in e_init delta_t f ==
```

```
let
  relax pi e s =
    let
      pi_new = f e beta
      delta_pi = sparse (pi_new-pi)
      delta_b = forall(i,j)->delta_pi!j
      delta_s = forall i->sum forall j->(w*delta_b)!(i,j)
      s_new = s+delta_s
      e_new = e+delta_t*(s_new-e)/tau_e
    in
      if converged pi pi_new then pi_new
      else
        relax pi_new e_new s_new
      in
        relax pi_in e_init
          beta+forall i->sum forall j->(w!(i,j)*pi_in!j)
```

Here, `beta`, `pi`, `E`, `s` and related entities are data fields of type `Df` τ `Float` with finite bounds, `w`: `Df` (τ, τ) `Float`, and `delta_s` is a data field with bound b whose elements `delta_s!i` are sums over the sparse data fields `forall j->(w*delta_b)!(i,j)`. The contribution from `delta_s` is then added into `s` and a new activity level is computed. The initial value of `s` is computed as a data field of sums in the "`j`"-direction.

The code above presumes that the weight matrix `w` is a dense data field, i.e., that it has dense bounds. But neural networks are often sparsely connected. Then most entries in `w` will be zero, and it can be efficient to turn `w` into a data field with sparse product bound (b_1, b_2) instead. The change to the code will be minor. `delta_s` will now have bound b_1 rather than b, and the line computing `s_new` must be changed into `s_new = s + fill delta_s 0` where `fill` is defined as

```
fill d c = forall x->if isoub d!x then c else d!x
```

With this change, the code will work as before but with a sparse weight matrix.

9 Related work

There are a number of functional formalisms which can be used as an abstract programming notation, to specify computations with indexed data structures on a mathematical level, or to identify and prove algebraic laws which can be used for program transformations.

An early example is Backus' FP [2]. FP is a formalism entirely based on functions and operations on functions, most prominently function composition. Values in FP are either *atoms* or *tuples* (sequences): the latter can be seen as (implicitly) indexed data structures.

Similar to FP in spirit is the *Bird-Meertens formalism* (BMF). Here an algebra with unary and binary functions forms a base for a set of theories for different data types [4, 5]. In particular, there is a theory for functions over lists. This formalism was originally developed to support the formal calculation of programs from specifications, but it can also serve as an abstract model for explicit data parallel programming, with a cost model [60]. The systematic transformation in BMF of specifications into parallel algorithms has been studied by Gorlatch [21]. Formal parallelisation in BMF is achieved by transforming functions into *list homomorphisms*: such functions can be computed in parallel in logarithmic time. A potential weakness of BMF in this context is that lists sometimes do not provide the best collection-oriented data type for modelling computational problems: in many

cases, explicitly indexed and possibly multidimensional structures are much more natural. Data fields are designed to express multidimensional problems well, and an interesting topic for future research is to investigate whether a corresponding concept of "data field homomorphism" can be developed and used in a similar way as the list homomorphisms in BMF.

A more machine-oriented, explicitly parallel model is the *Bulk-Synchronous Parallel* model (BSP), originally proposed by Valiant [68]. In this model, computations are indexed by processor ID's and are divided into distinct computation and communication phases. This makes it possible to assign a fairly simple cost model to the BSP model. A formal BSP calculus (an enriched λ-calculus) has been developed by Loulergue et. al. [48]: this calculus is different from our φ-calculi in that indices are implicit. There have also been attempts to apply a stepwise refinement program development methodology to the BSP model [61]. The BSP model could be seen as a restricted instance of the data field model, and it could serve as a target format for transformations from less restricted data field instances.

Algorithmic skeletons [11] are higher order patterns which can be used to implement collection-oriented operations in a way tailored to suit given parallel architectures. They thus fit very well in a functional context and it is not hard to imagine an implementation strategy for data fields where certain operations on data fields are translated into skeletal code.

There is some work on the formal modelling of arrays. The *Array Theory* [50] of Trenchard More, Jr. is an attempt to define an axiomatic theory of APL arrays. A virtue of this theory is the consistent handling of singularities, like empty arrays. This model is highly APL-specific and it is not easily generalized to other indexed structures. Another approach to arrays is taken by Fitzpatrick et. al. [18]: they define arrays as pairs of "shapes" and functions (essentially a simple instance of data fields) and consider algebraic transformations of high-level functional array algorithm specifications into forms suitable to implement on SIMD processors.

Formal models of arrays as *functions* from "index sets" are closely related to the partial function view of data fields. A Mathematics of Arrays (MOA) [51] is a model where APL-like arrays are defined in this manner, and operations on arrays are formally defined. In [9] *space-time recursion equations* are studied, a kind of recursively defined partial functions which give semantics to systolic arrays. This work later developed into the language Crystal [10, 71], where array-like entities called *data fields* are defined as functions ranging over finite *index domains*. These domains are constructed from a number of finite base domains (integer intervals, hypercube coordinates, trees) which can be combined using constructors for product, direct sum, function space, and restriction. The functions are explicitly typed with their index domains. A similar language is Alpha [42], where restrictions which ensure the efficient compilation are posed on the definition of recursive arrays [54, 55].

A similar view, but for more unstructured data, appears in the data parallel language Connection Machine Lisp [62]. 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 over a finite domain and they correspond to sparse data fields. A formalization of the Connection Machine Lisp model was done by Bougé and Paulin-Mohring [6].

These formalisms are all based on a single type of indexed data structure, and a purpose of our work is to develop a more generic model which is less dependent on the choice of structure. The work on *Shape Polymorphism* [34] is a step in that direction, where a category-theoretical model is used to specify *shapely types* and identify operations which are polymorphic over these types. The "usual" algebraic types of lists, trees, graphs etc. are shapely types. The canonical example of a shape-polymorphic function is `map`. The array language FiSh [35] is based on this theory. FiSh only supports regular arrays. Each array has a shape (similar to

bounds for data fields), and FiSh has fairly strong restrictions which ensure that the shape can be inferred at compile time. FiSh functions can be shape-polymorphic. A somewhat similar approach is the *polytypic* model, where functions like `map` are defined for classes of recursive types [33].

Another approach is to use systematic overloading to obtain a generic collection-oriented programming model. Peyton Jones [36] uses the class system of Haskell to define a class of *bulk types* with associated operations. Similar efforts have been done in the object-oriented community, like the STL C++ library [52]. Object-oriented approaches to generic program development for high-performance parallel computing have also been considered [14]. The *enum* function for data fields corresponds to the "iterator" design pattern in object-oriented design [19]. Common for these overloading-based efforts is that they do not pay much attention to multidimensional structures. It is possible to define a class for bounds in Haskell's current class system but multidimensional bounds cannot be handled well in this system [28].

Although the data field model is inspired by collection-oriented constructs in existing languages, it is still a formal model and therefore this section focusses on formalisms rather than languages. An excellent survey of collection-oriented languages and features up to around 1990 is found in [58]. However, the languages ZPL [8] and FIDIL [57] deserve special mentioning since they consider bounds (called *regions* in ZPL and *domains* in FIDIL) to be more or less first-class data. In particular, the domains of FIDIL are much richer than simple array bounds: they can also be sparse, or "boxed" (exact unions and set differences of array bounds). The data field model would be well suited to describe them formally. FIDIL is also the only existing array-like language we know which does not require conforming arguments for elementwise applied strict operations: rather, it uses the "implicit intersection rule" which corresponds to our equation (5) for computing bounds of φ-terms.

10 Conclusions and Further Research

We have presented the data field model, which is a extension of the traditional array concept into a general model for indexed data structures. A major aim is to provide semantically sound and general principles for how to design languages with collection-oriented features. Data fields are pairs of functions and general bounds equipped with a number of abstract operations satisfying some axioms. The operations are selected to support the usual collection-oriented operations.

Indexed structures can also be seen as partial functions. λ-abstraction turns out to provide a very flexible and generic syntax for defining many collection-oriented operations on partial functions. We define φ-abstraction as a similar syntax for data fields. The purpose is twofold: it supports a style of defining data fields as if they were partial functions, and its gives a way to define data fields without explicitly defining the bounds. The bounds are instead implicitly defined through the semantics of φ-abstraction. We give a family of possible semantics for φ-expressions as higher order rewrite systems. They define how bounds should be inferred in some cases which occur to some extent in existing languages: elementwise applied operations, selection and projection on higher-dimensional arrays, and indirect indexing. We prove a number of properties for the semantics of φ-expressions, including a theorem which relates the semantics of a λ-expression and the corresponding φ-expression. This is, however, not the only possible way to define bounds implicitly, and we outline an alternative φ-calculus which models the more conventional "conformance requirement" for operands to elementwise applied operations. We also consider briefly how mutable data fields, which can be updated in-place, could be formally defined.

We define a small functional language with data fields and `forall`-abstraction

which we use for programming and language design examples throughout the text. The data field part is given a semantics in terms of formal data fields and φ-abstraction. The language provides a possible core language for data fields, and we show how a number of syntactical conveniences can be built on top of it. The program examples include database queries, symbolic representations of drawings, and parallel algorithms for arrays and sparse structures. Our aim is to prove that data fields can provide a suitable programming concept for a wide range of applications, including but not restricted to parallel algorithms and their specification. On the other hand it is also possible to define application-specific languages based on specialised data fields, where the language features and the exact data field semantics are tuned to provide the best tradeoff for the application at hand between efficiency, expressiveness and flexibility.

We have implemented a dialect of Haskell which offers an instance of data fields [27]. This dialect provides data fields with sparse/dense array bounds almost exactly as defined in Sect. 7.5, and several of the examples given here have been implemented in it. One topic of research for the future is to develop this dialect further, both regarding language features and implementations, and to try it out in a more varied range of applications.

Another topic of research is how to integrate the elemental intrinsics overloading with more advanced type systems than the simple, explicitly typed system of our example language. It is not known how type systems with type inference á la Hindley-Milner should be best modified to accomodate this. In Haskell, the class system could be used to some extent to provide this overloading but it does not seem to provide the best way of doing it. We are currently working on a modified Hindley-Milner type system which resolves this overloading at compile-time [66].

This presentation has focussed on the language design aspects of the data field model, but it could also potentially act as a foundation for formal program development methodologies. We have already mentioned the possible connection with parallel algorithm development in the Bird-Meertens formalism. Another, fairly obvious use of the model which we have not developed here is as a framework for studying formal mappings of data structures, e.g., from abstract index domains close to the problem to concrete index domains close to the address space of the target machine.

Acknowledgments

We would like to thank Jan-Olof Eklundh, Karl-Filip Faxén, Jonas Holmerin, Fredrik Nöu, and Claes Thornberg for valuable comments. This work was partially supported by The Swedish Research Council for Engineering Sciences (TFR), grants 91–333 and 94–109. Part of the work was done while the second author was Invited Professor at Ecole Normale Supérieure de Lyon.

References

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

[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] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, 1985.

[4] 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.

[5] 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.

[6] L. Bougé and C. Paulin-Mohring. Towards a theory of data-parallel computation: from Connection Machine Lisp to Connection Machine ML. Unpublished draft, Mar. 1991.

[7] W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. Programming Languages. McGraw-Hill, 1990.

[8] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, 1998.

[9] M. C. Chen. *Space-Time Algorithms: Semantics and Methodology*. PhD thesis, California Institute of Technology, Pasadena, CA, 1983.

[10] 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.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[12] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9, pages 459–492. Elsevier Science Publishers B. V., 1990.

[13] G. Dahlquist, Å. Björck, and N. Anderson. *Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ, 1974.

[14] F. Dobrian, G. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. ICASE Report 99-38, ICASE, Hampton, VA, Sept. 1999.

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

[16] A. Falkoff and K. Iverson. The Design of APL. *IBM Journal of Research and Development*, pages 324–333, July 1973.

[17] 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.

[18] S. Fitzpatrick, T. J. Harmer, A. Stewart, M. Clint, and J. M. Boyle. The automated transformation of abstract specifications of numerical algorithms into efficient array processor implementations. *Science of Computer Programming*, 28(1):1–41, 1997.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Professional Computing Series. Addison-Wesley Longman, 1995.

[20] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. Assoc. Comput. Mach.*, 24(1):68–95, Jan. 1977.

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

[22] P. Hammarlund and A. Lansner. Implementations of very large recurrent ANNs on massively parallel SIMD computers. In I. Aleksander and J. Taylor, editors, *Proceedings of the 1992 International Conference on Artificial Neural Networks*, pages 1287–1290, Amsterdam, September 1992. ICANN-92, North-Holland.

[23] 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.

[24] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1):1–170, June 1993. HPF Version 1.0.

[25] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, Dec. 1986.

[26] J. R. Hindley. *The Church-Rosser Property and a Result in Combinatory Logic*. PhD thesis, University of Newcastle-upon-Tyne, 1964.

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

[28] J. Holmerin and B. Lisper. Data Field Haskell. Unpublished draft, 1999.

[29] J. Holmerin and B. Lisper. Data Field Haskell. In G. Hutton, editor, *Proc. Fourth Haskell Workshop*, pages 106–117, Montreal, Canada, Sept. 2000.

[30] J. Holmerin and B. Lisper. Development of parallel algorithms in Data Field Haskell. In A. Bode, T. Ludwig, W. Karl, and R. Weismüller, editors, *Proc. Euro-Par 2000*, volume 1900 of *Lecture Notes in Comput. Sci.*, pages 762–766, Munich, Germany, Aug. 2000. Springer-Verlag.

[31] P. Hudak, S. Peyton 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.

[32] J. Hughes. Lazy memo-functions. In J.-P. Jouannaud, editor, *Proc. Functional Programming Languages and Computer Architecture*, pages 129–146, Nancy, France, Sept. 1985. Springer-Verlag.

[33] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, Jan. 1997. ACM Press.

[34] C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Proc. 5th European Symposium on Programming*, Volume 788 of *Lecture Notes in Comput. Sci.*, pages 302–316, Edinburgh, Apr. 1994. Springer-Verlag.

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

[36] S. P. Jones. Bulk types with class. In *Electronic Proceedings of the 1996 Glasgow Functional Programming Workshop*, Ullapool, July 1996.

[37] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. Mathematical Centre Tracts Nr. 127.

[38] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, vol. 2*, chapter 1, pages 1–116. Oxford University Press, Oxford, 1992.

[39] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoret. Comput. Sci.*, 121:279–308, 1993.

[40] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y., 1970.

[41] A. Lansner and Ö. Ekeberg. A one-layer feedback, artificial neural network with a Bayesian learning rule. *Int. J. Neural Systems*, 1(1):77–87, 1989.

[42] H. Le Verge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. *Journal of VLSI Signal Processing*, 3:173–182, 1991. 1991.

[43] B. Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Vol. 1132 of *Lecture Notes in Comput. Sci.*, pages 220–251, Les Ménuires, France, Mar. 1996. Springer-Verlag.

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

[45] 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.

[46] B. Lisper and P. Hammarlund. The data field model. Technical Report TRITA-IT R 99:02, Dept. of Teleinformatics, KTH, Stockholm, Mar. 1999.
`ftp://ftp.it.kth.se/Reports/TELEINFORMATICS/TRITA-IT-9902.ps.gz`.

[47] B. Lisper and J. Holmerin. Development and verification of parallel algorithms in the data field model. In S. Gorlatch and C. Lengauer, editors, *Proc. 2nd Int. Workshop on Constructive Methods for Parallel Programming*, pages 115–130, Ponte de Lima, Portugal, July 2000.

[48] F. Loulergue, G. Hains, and C. Foisy. A calculus of recursive-parallel BSP programs. In S. Gorlatch, editor, *Proc. First International Workshop on Constructive Methods for Parallel Computing*, pages 59–70, Marstrand, Sweden, June 1998.

[49] D. Maier. *The Theory of Relational Databases*. Pitman, London, 1983.

[50] T. More, Jr. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development*, 17(2):135–175, Mar. 1973.

[51] L. M. R. Mullin. Psi, the indexing function: a basis for FFP with arrays. In L. M. R. Mullin, M. Jenkins, G. Hains, R. Bernecky, and G. Gao, editors, *Arrays, Functional Languages, and Parallel Systems*, chapter 10, pages 185–200. Kluwer Academic Publishers, Boston, 1991.

[52] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, 1996.

[53] E. Part-Enander, A. Sjöberg, B. Melin, and P. Isaksson. *The MATLAB Handbook*. Addison-Wesley, 1996.

[54] 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.

[55] 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.

[56] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *J. Assoc. Comput. Mach.*, 27(4):772–796, Oct. 1980.

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

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

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

[60] D. B. Skillicorn. Architecture-independent parallel computation. *Computer*, 23(12):38–50, Dec. 1990.

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

[62] 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.

[63] V. Strassen. Gaussian elimination is not optimal. *Numerical Mathematics*, (13), 1969.

[64] Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in C**, 6.1 edition, 1991.

[65] Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in *Lisp*, 6.1 edition, 1991.

[66] C. Thornberg. *Towards Polymorphic Type Inference with Elemental Function Overloading.* Licentiate thesis, Dept. of Teleinformatics, KTH, Stockholm, May 1999. Research Report TRITA-IT R 99:03.

[67] 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.

[68] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, Aug. 1990.

[69] V. van Oostrom. Private communication.

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

[71] 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.

A A Rewrite Semantics for the Metalanguage in Section 4

The following CRS M gives an alternative semantics to the metalanguage defined in Sect. 4, including the "hyperstrictification" operator defined in Sect. 4.2. (For readability the notation is not in formal CRS syntax, but it is straightforward to put the rules in this format. See Appendix B.)

$$op(c_1,\ldots,c_n) \to c, \tag{37}$$
$$is_*(*) \to true \tag{38}$$
$$is_*(c) \to false \tag{39}$$
$$if\,(true,x,y) \to x \tag{40}$$
$$if\,(false,x,y) \to y \tag{41}$$
$$if\,(*,x,y) \to * \tag{42}$$
$$\lambda x.t\,t' \to t[t'/x] \tag{43}$$
$$\mu f.t \to t[\mu f.t/f] \tag{44}$$
$$\overline{f}\,c \to f\,c, \quad c \text{ normal form with no occurrence of } * \tag{45}$$
$$\overline{f}\,c \to *, \quad c \text{ normal form with some occurrence of } * \tag{46}$$

($t[t'/x]$ denotes substitution with t' for x in t.) We call this CRS M. (37) is really a rule scheme with an instance for every strict operation op, possible tuple $(c_1,\ldots,c_n)$ of normal forms for which f is defined, and resulting function value c. (This is a standard way to specify strictness in rewrite systems. Cf. the eager λ-calculus [70].) (45) and (46) are also rule schemes of this kind. Similarly, (39) is a rule scheme, where c ranges over an assumed set of nonoverlapping patterns which together match exactly the possible canonical forms corresponding to non-$(\bot,*)$ values in the cpo which is_* maps from. (We omit the details. Just to give the flavour, for a lazy language where expressions are reduced to weak head normal form the patterns are of the form $C(x_1,\ldots,x_n)$, for each constructor C of arity $n \geq 0$.)

The other rules except (42) are more or less the same as the reduction rules of PCF [3]. The correspondence between least fixpoint and rewrite models for this kind of language is well known [3, 12, 56]. Let us just make two remarks: first, M is left-normal and orthogonal, which means that it is confluent and the leftmost-outermost reduction strategy is normalizing (see Appendix B). Second, if $t \leftrightarrow^* t'$ where t, t' are terms in the metalanguage and $\leftrightarrow^*$ is the convertibility relation of M, then t and t' must be equal also w.r.t. the denotational semantics.

B Combinatory Reduction Systems

Combinatory Reduction Systems (CRS) is a generalization of (first order) term rewriting systems (TRS) which includes reduction systems with bound variables, like different λ-calculi. Many important concepts and results for TRS, such as orthogonality, and results about reduction strategies, carry over directly to CRS. For a full description, including fully formal definitions, see [37, 39].

Consider a set of terms T, constructed out of constant function symbols with fixed arity, nullary variables, and a binary *abstraction operator*, written $[\_]\_$ (i.e., if t is a term and x is a variable, then $[x]t$ is a term). A Combinatory Reduction System over T is a set of *reduction rules* $s \to t$, where s, t are *metaterms*, constructed as the terms in T, plus terms containing *metavariables*, written in upper case (Z, Z', etc.). Each metavariable has an arity (possibly 0): if Z has arity k, then $Z(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are metaterms, is a metaterm. The metavariables correspond to the free variables in TRS rules.

(Meta)terms are considered equivalent modulo renaming of bound variables. For metaterms s, t in reduction rules $s \to t$, we demand (i) s and t are closed (i.e., a variable x occurs only within the scope of a binding $[x]\_$), (ii) s has the form $F(t_1, \ldots, t_n)$, where F is a constant function symbol, (iii) any metavariable in t occurs also in s, and (iv) a metavariable Z of arity k occurs in s only in the form $Z(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are pairwise distinct variables. Examples are the CRS rules for β-reduction and unfolding of fixpoint abstractions:

$$\begin{aligned} @(\lambda([x]Z(x)), Z') &\to Z(Z') \\ \mu x.Z(x) &\to Z(\mu x.Z(x)) \end{aligned}$$

("@" is a binary function symbol for application.)

A CRS R generates a reduction relation $\to_R$ on the set of terms, very much like a first order TRS. The main difference is that the matching of rule to subterm is performed by a *valuation* rather than a first order substitution. Valuations are essentially a complex form of substitution which allows the possibility to specify syntactically in rules whether bound variables can occur in certain subterms or not. The only way for a bound variable to occur in a subterm matched to a metavariable is if it is explicitly "passed" as a formal argument to the metavariable, i.e., a term which is matched by $[x](\ldots Z \ldots)$ (where Z is a nullary metavariable) can have no occurrence of x in the subterm corresponding to Z, whereas a term matched by $[x](\ldots Z(x) \ldots)$ (with Z unary) may have this. This allows purely syntactical specifications of rules like η-reduction in the λ-calculus.

Two important concepts for CRS are *orthogonality* and *left-normality*. Two rewrite rules are mutually orthogonal if both are left-linear and if they don't overlap [39] (i.e., there are no "critical pairs" in the terminology of Knuth-Bendix completion [38, 40]). A CRS is orthogonal if all its rules are mutually orthogonal. An orthogonal CRS is confluent [37, 39]. A CRS is left-normal if, in the left-hand sides of all rules, all constants and function symbols (in linear term notation) precede the variables and metavariables. For a left-normal CRS the leftmost-outermost reduction strategy is normalising [37].

Most facts about CRS still hold for substructures, i.e., systems where the set of terms is restricted but closed under reduction, valuation, and taking of context. Such substructures include typed systems. In particular, the facts about orthogonal and left-normal systems mentioned above still hold.

C CRS rules for φ-abstraction

The reduction rules

$$\varphi x.t \to (\lambda x.t, B_i(t, x, \emptyset))$$

which define the CRS $\Phi_i(R)$ have the following formal CRS definition: for all terms t and variables x such that $B_i(t, x, \emptyset))$ is defined, there is a rule

$$\varphi([x]t) \to (\lambda([x]t), B_i(t, x, \emptyset))$$

The rules are particularly simple since they contain no metavariables. Rules

$$\varphi(x_1, \ldots, x_n).t \to (\lambda(x_1, \ldots, x_n).t, B_i(t, (x_1, \ldots, x_n), \emptyset))$$

with pattern-matching over tuples have the following formal CRS definition:

$$\varphi([x_1] \cdots [x_n]t) \to (\lambda([x_1] \cdots [x_n]t), B(t, (x_1, \ldots, x_n), \emptyset))$$

D Proofs of properties of $\Phi_i(R)$

Proof of Proposition 8. $B_i(t, x, Y)$ is defined only if $FV(t) \subseteq \{x\} \cup Y$. Thus, $\varphi x.t \to (\lambda x.t, B_i(t, x, \emptyset))$ belongs to $\Phi_i(R)$ only if $\varphi x.t$ is closed. Therefore $\Phi_i(R)$ is trivially left-linear, since all left-hand sides are closed. Furthermore, $B_i(t, x, Y)$ is defined only if t contains no closed φ-subterm. Since all left-hand sides in $\Phi_i(R)$ are closed φ-terms, it follows that there can be no overlap between them. ■

Proof of Proposition 9. It remains to check that there is no overlap between rules in $\Phi_i(R)$ and R. No rule of $\Phi_i(R)$ can match any subterm in a left-hand side of a rule in R, since these cannot have any subterms of form $\varphi x.t$. Conversely, no rule in R can match any subterm in a left-hand side of a rule in $\Phi_i(R)$, since any such left-hand side is an R-nf. ■

Proof of Proposition 10. The reduction relations of two mutually orthogonal CRS'es commute [56, 69]. $\Phi_i(R)$ is orthogonal, thus confluent. Then Hindley-Rosen's lemma [26] yields that the union is confluent. ■

Proof of Proposition 11. Consider any closed $\Phi_i(R) \cup R$-nf t. Either t contains no closed φ-subterms. But then, since t is closed, any φ-subterm contains some variable each which is bound by some other abstraction mechanism than φ. t is then allowed to be a $\Phi_i(R) \cup R$-nf. Otherwise, t must have some closed subterm $\varphi x.t'$. Let us show, by contradiction, that t then cannot be a $\Phi_i(R) \cup R$-nf. Since t is a $\Phi_i(R) \cup R$-nf, t' must be an R-nf. If all its φ-subterms are open, then $B_i(t', x, \emptyset)$ is defined: thus, $\varphi x.t'$ is a $\Phi_i(R)$-redex and t cannot be a $\Phi_i(R) \cup R$-nf. Otherwise t' has some closed φ-subterm. We can now recursively test this subterm in the same way, until we either arrive at a subterm which has only open φ-subterms, and then is a $\Phi_i(R)$-redex, or an innermost closed φ-subterm without any φ-subterm. This innermost subterm must then be a $\Phi_i(R)$-redex. ■

E Proofs of Theorem 1 and Related Lemmas

The following two lemmas are variations on the results in Section 6. We need them to prove Lemma 2. Their proofs are entirely straightforward.

Lemma 4 *If g is strict, then $res(g(f_1,\ldots,f_n)) \subseteq res(f_1) \cap \cdots \cap res(f_n)$.*

Lemma 5 *$res(if(b,f,g)) \subseteq res(b) \cap (res(f) \cup res(g))$.*

Proof of Lemma 1. First note that for any predicate p it holds that $\{\, x \mid \overline{p}(x) = true \,\} \subseteq \{\, x \mid p(x) = true \,\}$. We have $res(\lambda x.(f,b)\,!\,g(x)) = res(\lambda x.\overline{f \setminus b}(g(x)) =$ (by Proposition 1) $= res(\lambda x.(\overline{f}\setminus\overline{b})(g(x)) =$ (by Proposition 3) $= res(\lambda x.\overline{f}(g(x))\setminus \overline{b}(g(x)) \subseteq \{\, x \mid \overline{[\![b]\!]}(g(x)) = true \,\} \subseteq \{\, x \mid [\![b]\!](g(x)) = true \,\} = \{\{[\![b]\!] \circ g\}\}$. ■

Proof of Lemma 2. By induction on expressions, considering each of the equations (2) – (11) defining $B_0(t,x,Y)$:

- $B_0(t,x,Y) = all$: then the statement is trivially true. This case includes the base cases $t = c$ and $t = y \in \{x\} \cup Y$.
- $t = (f,b)!x$, $FV(f,b) = \emptyset$: then $B_0(t,x,Y) = b$ and $res(\lambda x.t^v) = res(\lambda x.((f,b)!x)) \subseteq \{\{b\}\}$ by Lemma 1. By the assumed property of $\mathcal{B}(\alpha)$ there exists a maximal bound b' such that $B_0(t,x,Y) = b \sqsubseteq b'$. Thus, $res(\lambda x.t^v) \subseteq \{\{b'\}\}$, and by Proposition 7 we also obtain $\{\{B_0(t,x,Y)\}\} \subseteq \{\{b'\}\}$.
- $t = (f,b)\,!\,t_1$, $FV(f,b) = \emptyset$, $t_1 \neq x$: induction on t_1. Since $FV(f,b) = \emptyset$ we have $t^v = \lambda x.(f,b)\,!\,t_1^v$. We have $z \in res(\lambda x.(f,b)\,!\,t_1^v) \implies (\lambda x.(f,b)\,!\,t_1^v)\,z \neq \bot \implies (f,b)!t_1^v[z/x] \neq \bot \implies \overline{f \setminus b}(t_1^v[z/x]) \neq \bot \implies (\overline{f \setminus b}$ hyperstrict$) \implies t_1^v[z/x] \neq \bot \implies z \in res(\lambda x.t_1^v)$. Thus, $res(\lambda x.t^v) \subseteq res(\lambda x.t_1^v)$. By the induction hypothesis there is a maximal b' such that $res(\lambda x.t_1^v) \subseteq \{\{b'\}\}$ and $\{\{B_0(t_1,x,Y)\}\} \sqsubseteq b'$. Since $\{\{B_0(t,x,Y)\}\} = \{\{B_0(t_1,x,Y)\}\}$ the result follows for t.
- $t = op(t_1,\ldots,t_m)$, op strict: then $B_0(t,x,Y) = B_0(t_1,x,Y) \sqcap \cdots \sqcap B_0(t_m,x,Y)$. We have $t^v = op(t_1^v,\ldots,t_m^v)$. By induction, there are maximal bounds b_i such that $res(\lambda x.t_i^v) \subseteq \{\{b_i\}\}$ and $B_0(t_i,x,Y) \sqsubseteq b_i$ for $1 \leq i \leq n$. Finally, note that Definition 4 requires that $\{\{b\}\} \cap \{\{b'\}\} \subseteq \{\{b \sqcap b'\}\}$ for all maximal bounds b, b'. We have $res(\lambda x.t^v) = res(\lambda x.op(t_1^v,\ldots,t_m^v)) = res(op(\lambda x.t_1^v,\ldots,\lambda x.t_m^v)) \subseteq$ (by Lemma 4) $\subseteq res(\lambda x.t_1^v) \cap \cdots \cap res(\lambda x.t_n^v) \subseteq$ (by induction) $\subseteq \{\{b_1\}\} \cap \cdots \cap \{\{b_m\}\} \subseteq \{\{b_1 \sqcap \cdots \sqcap b_m\}\}$. Since all b_i are maximal it holds that $b_1 \sqcap \cdots \sqcap b_m$ is maximal. Furthermore, $B_0(t,x,Y) = B_0(t_1,x,Y) \sqcap \cdots \sqcap B_0(t_m,x,Y) \sqsubseteq$ (by induction plus monotonicity of $\sqcap$) $\sqsubseteq b_1 \sqcap \cdots \sqcap b_m$. Thus, $b_1 \sqcap \cdots \sqcap b_m$ has the desired properties.
- $t = if(t_1,t_2,t_3)$: analogous to the previous case.
- $t = \lambda y.t_1$: the induction hypothesis is $\forall Y \forall v.\exists b.res(\lambda x.t_1^v) \subseteq \{\{b\}\} \wedge B_0(t_1,x,Y) \sqsubseteq b$, where b is maximal. The first conjunct can be reformulated into $\forall Y \forall v \exists b.(\forall z.z \in res(\lambda x.t_1^v) \implies [\![b]\!]\,z)$, or $\forall Y \forall v \exists b.(\forall z.t_1^v[z/x] \neq \bot \implies [\![b]\!]\,z)$, or $\forall Y \forall v \exists b.(\forall z.t_1^v[z/x] = \bot \vee [\![b]\!]\,z)$. Now, we may replace Y with $Y \cup \{y\}$ and v with $v \cup w$ in the induction hypothesis, where v maps from Y to values and w from $\{y\}$ to values. Also note that $t_1^{v \cup w} = (t_1^v)^w$. We obtain $\forall Y \forall y \forall v \forall w \exists b.(\forall z.(t_1^v)^w[z/x] = \bot \vee [\![b]\!]\,z) \wedge B_0(t_1,x,Y \cup \{y\}) \sqsubseteq b$. The second conjunct equals $B_0(\lambda y.t_1,x,Y) \sqsubseteq b$. Now we want to move the quantification over w inside the existential quantification over b. In order to do this, we observe that in general it holds that $\forall x.\exists y.(P(x) \vee Q(y)) \wedge R(y) \iff \exists y.(\forall x.P(x) \vee Q(y)) \wedge R(y)$, which

is straightforward to prove. Thus, we are allowed to move the quantification over w into the first disjunct, which becomes $\forall w.(t_1^v)^w[z/x] = \bot$. Due to extensionality, this is the same as $\lambda y.t_1^v[z/x] = \bot$. Thus, we obtain $\forall Y \forall y \forall v \exists b.(\forall z.\lambda y.t_1^v[z/x] = \bot \vee [\![b]\!]\, z) \wedge B_0(\lambda y.t_1, x, Y) \sqsubseteq b$, or $\forall Y \forall v \exists b.(\forall z.z \notin res(\lambda x.\lambda y.t_1^v) \vee [\![b]\!]\, z) \wedge B_0(\lambda y.t_1, x, Y) \sqsubseteq b$, or
$\forall Y \forall v \exists b.(res(\lambda x.\lambda y.t_1^v) \subseteq [\![b]\!]) \wedge B_0(\lambda y.t_1, x, Y) \sqsubseteq b$.

- $t = \varphi y.t_1$: We need the following lemma: $\forall v.t_v = \bot_\beta \implies \varphi x.t = \bot_{\mathcal{D}(\alpha,\beta)}$. From this, the proof can be carried out exactly as in the previous case.

  We now prove the lemma. There are two cases. First, assume t has no normal form. Then directly $\varphi x.t = \bot_{\mathcal{D}(\alpha,\beta)}$. Otherwise, t has a normal form t_1. Then $\varphi x.t = (\lambda x.t_1, B_0(t_1, x, \emptyset))$. Since rewrite systems are closed under substitutions and $t \to^* t_1$ we obtain $t^v \to^* t_1^v$. This means that $t^v = t_1^v$ when their meanings in the metalanguage are considered, since this meaning must be consistent with the rewrite semantics for the language. Thus, $t_1^v = \bot_\beta$, and $\varphi x.t = (\lambda x.\bot_\beta, B_0(t_1, x, \emptyset))$. Thus, $[\![\varphi x.t]\!] = \bot_{\alpha\to\beta}$ which, by extensionality of data fields, implies that $\varphi x.t = \bot_{\mathcal{D}(\alpha,\beta)}$.

■

Proof of Theorem 1. Let t be as stated in the theorem and let y be finite maximal. We have $\varphi x.t\,!\,y = \overline{\lambda x.t' \setminus [\![B_0(t', x, \emptyset)]\!]}\, y =$ (since y finite maximal) $= \lambda x.t' \setminus [\![B_0(t', x, \emptyset)]\!]\, y$. Now, if $y \in res(\lambda x.t')$, then, by Lemma 2 follows that there is a maximal b such that $[\![b]\!]\, y = true$ and $B_0(t', x, \emptyset) \sqsubseteq b$. Since $B_0(t', x, \emptyset)$ is maximal it must then be equal to b, and it follows that $[\![B_0(t', x, \emptyset)]\!]\, y = true$. Thus, $\varphi x.t\,!\,y$ equals $\lambda x.t'\, y$. If, on the other hand, $y \notin res(\lambda x.t')$, then $\lambda x.t'\, y = \bot$, and thus also $\varphi x.t'\,!\,y = \bot$.

The above proves that $\varphi x.t\,!\,y = \lambda x.t'\, y$. But since t reduces to t' we must have $\lambda x.t'\, y = \lambda x.t\, y$, since the meaning of the λ-abstraction must be consistent with the rewrite semantics for the metalanguage (cf. the proof of Lemma 2). Thus, the theorem is proved. ■

Proof of Lemma 3. For all the cases where B_0 is defined, the proof of Lemma 2 carries over more or less directly. The new, nontrivial case, is $t = (f, b)\,!\,\vec{t}[p, \vec{x}]$, where $\emptyset \subset FV(t_i) \subseteq Y$ for $i \in I$ and $FV(t_i) = \emptyset$ for $i \notin I$. Then $t^v = (f, b)\,!\,\vec{t}^v[p, \vec{x}]$ and $B_1(t, \vec{x}, Y) = bproj_{p,I}(b, \vec{t})$. By Lemma 1, $res(\lambda\vec{x}.(f, b)\,!\,\vec{t}^v[p, \vec{x}]) \subseteq \{\, \vec{x} \mid [\![b]\!](\vec{t}^v[p, \vec{x}]) = true \,\}$. Set $\vec{y} = \vec{t}^v$. Thus, for any $\vec{x}$ in this set, $\exists(\, y_i \mid i \in I \cup dom(p)\,).[\![b]\!](\vec{y}[p, \vec{x}]) = true$. By (12), this implies that $[\![bproj_{p,I}(b, \vec{y})]\!](\vec{x}) = true$. Now $t_i^v = t_i$ for $i \notin I$, since these t_i are closed. Thus, $t_i = y_i$ for these i. It then follows from (12) that also $[\![bproj_{p,I}(b, \vec{t})]\!](\vec{x}) = true$. (For this, it suffices that $t_i = y_i$ for $i \notin I \cup dom(p)$.) Now, precisely as in the case $t = (f, b)\,!\,x$ in the proof of Lemma 2, the existence of a maximal b' such that $res(\lambda\vec{x}.t^v) \sqsubseteq b'$ and $\{\{B_1(t, \vec{x}, Y)\}\} \subseteq \{\{b'\}\}$ follows. ■