Extent Analysis of Data Fields

Björn Lisper[1] * and Jean-Francois Collard[2] **

[1] Department of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, SWEDEN
[2] LIP, ENS Lyon, 46 Allée d'Italie, 69364 LYON CEDEX 07, FRANCE

Abstract. Data parallelism means operating on distributed tables, *data fields*, in parallel. An abstract model of data parallelism, developed in [10], treats data fields as functions explicitly restricted to a finite set. Data parallel functional languages based on this view would reach a very high level of abstraction. Here we consider two static analyses that, when successful, give information about the *extent* of a data field with recursively defined elements, in the form of a predicate that is true wherever the data field is defined. This information can be used to preallocate the elements and map them efficiently to distributed memory, and to aid the static scheduling of operations. The predicates can be seen as extensions, providing more detail, of classical data dependency notions like strictness. The analyses are cast in the framework of abstract interpretation: a forward analysis which propagates restrictions on inputs to restrictions on outputs, and a backward analysis which propagates restrictions the other way. For both analyses, fixpoint iteration can sometimes be used to solve the equations that arise. In particular, when the predicates are linear inequalities, integer linear programming methods can be used to detect termination of fixpoint iteration.
Keywords: data parallelism, functional programming, abstract interpretation, arrays.

1 Introduction

Data parallelism [11] is a programming paradigm where operations are made in parallel over a "data field". The term was first coined by Yang and Choo [19]; we adopt it since it is not tied to any specific "data parallel data type". Arrays are examples of data fields, but the concept can also cover more unstructured collections of data. Typical operations on data fields are: pointwise applied "scalar" operations, "reduction" (sums, products and similar), "scan" or "parallel prefix" operations (forming a data field of all partial "sums"), and various permutations and rearranging operations that can be interpreted as communication operations. Data parallelism usually maps well to SIMD parallel and pipelined vector

* lisper@it.kth.se. Partly supported by The Swedish Research Council for Engineering Sciences (TFR), grant no. 94-109.
** Jean-Francois.Collard@ens-lyon.fr. Partly supported by the French CNRS Program C^3, and DRET contract 91/1180.

computers, and is therefore often considered an explicit programming method for such machines. It is, however, often a surprisingly convenient paradigm for expressing algorithms in several areas, like neural network computations, computational fluid dynamics, and linear algebra. This indicates that data parallelism should be considered an *abstract programming paradigm* rather than a form of explicit parallel programming of a class of computer architectures. Especially, data parallelism has been found to be a suitable paradigm for programming massively parallel, data- and computation-intensive applications, which do not necessarily have a regular structure.

In [10], the approach was taken to consider data fields as functions over finite sets, tabulated in a distributed fashion. Operations on data fields are then simply second order operations on functions. All commonly occurring data parallel operations have side-effect free forms that are conveniently expressed in this way, see [10]. A higher order functional language can be extended with a single second order operation for *explicit restriction* of functions to certain arguments. If a restricted function is defined in only a finite number of points, it can then be represented by a table with an entry for each argument where it is defined. A higher-order functional language extended with explicit restriction can thus be seen as a data parallel functional language. Such a language will enjoy all the good software engineering properties of declarative languages, and it can be made small, abstract, general, and elegant. For a lazy functional language extended in this way, recursive programs will have straightforward denotational semantics given by the least fixpoint solutions of their recursive definitions.

The purpose of this paper is to describe a framework for analysis of data fields defined as functions with explicit restrictions. The aim is (1) for a given recursive definition of a data field, to infer the points where it is possibly defined, and (2), given a function call that involves accesses to a data field, to infer where the data field is accessed (and thus has to be computed). We will refer to this as *extent analysis*. If extent analysis succeeds, the result can be used to allocate memory at compile time for the data field. On distributed memory machines we can then also map data field elements to physical processors, which is a crucial operation for performance on systems with highly nonuniform memory access times. Extent analysis will therefore be an interesting program analysis in a compiler for a functional data parallel language along the lines described above. Sciences

2 Data Fields

Following [10], we define a *data field* to be a function $f{:}C \rightarrow C'$, where C and C' are complete partial orders (cpo's). Furthermore, we define $dom(f) = \{\, x \mid f(x) \neq \bot \,\}$. Clearly, if we want to represent f as a *table*, it is sufficient to tabulate f only for the points in $dom(f)$, and consider the result of a table lookup undefined for any argument not in $dom(f)$.

Let B be the flat cpo of booleans $\{true, false, \bot_B\}$. For any cpo C with

bottom element $\perp_C$, we define the ternary function $if_C : B \times C \times C \to C$ by:

$$if_C(\perp_B, x, y) = \perp_C, \quad if_C(true, x, y) = x, \quad if_C(false, x, y) = y. \tag{1}$$

We will most often drop indices B, C etc. when these are clear from the context. We now define explicit "undefinition", or *where-restriction*, as follows, for any function $f : C \to C'$ and predicate $b : C \to B$:

$$f \setminus b = \lambda x.if(b(x), f(x), \perp).$$

This can be read "f where b" and it is a function $C \to C'$ that is explicitly undefined for all x such that $b(x) \neq true$. This single new construct turns a higher order functional language into a data parallel language. An example of its use is to make an array $A[1 \ldots n]$ out of a function $A : \mathbb{Z}_\perp \to C$ (where C is some cpo and $\mathbb{Z}_\perp$ is the flat cpo of integers):

$$A \setminus (\lambda i.1 \leq i \leq n).$$

Cf. array comprehensions in Haskell [1].

The "data fields as functions" approach is, however, not implementable in its general form: to decide where $f(x) \neq \perp$, or even whether $dom(f)$ is finite, amounts to solving the halting problem. For some combinations of data types and explicit where-restriction, finiteness is, however, decidable. Some examples are:

- Functions over a finite cpo (trivially).
- Functions $f \setminus b$, where f is a function from $\mathbb{Z}_\perp^n$ for some $n > 0$, and b consists of a number of disjunctions and/or conjunctions of linear inequalities. Finiteness can be decided by solving a number of integer linear programming problems. Interestingly, this can be done even when b contains symbolic parameters, e.g. through Feautrier's "Parametric Integer Programming" algorithm [7].

Thus, there are instances where this approach to data parallelism is implementable. Especially, it offers a semantically clean way to define arrays and operations on them. The approach is, however, not limited to arrays, but applies to other proposed carriers of data parallel entities as well, such as lists [16], nested finite sequences [2, 3], or the "xappings" in CM Lisp [17].

Operations on data fields can be expressed as second order operations on functions. *Elementwise application* of a scalar n:ary operation $g : C_1 \times \cdots \times C_n \to C'$ to n data fields $f_i : C \to C_i$, $1 \leq i \leq n$, is simply n-ary function composition of g with $f_1, \ldots, f_n$:

$$g(f_1, \ldots, f_n) = \lambda x.g(f_1(x), \ldots, f_n(x)).$$

This is a data field $C \to C'$. *Get communication* is also function composition, but with the data field to the left: if f is a data field $C' \to C$ and g a function $C'' \to C'$, then

$$f(g) = \lambda x.f(g(x))$$

is a data field $C'' \to C$ formed by fetching, for each x, the component of f given by $g(x)$. More complex operations, like *reduction* and *scan* (parallel prefix), can be defined in terms of these primitive operations and recursion. See [10].

Let us finally mention an identity (slightly adapted from [10, corollary 1]), that will be of importance here:

Proposition 1. *If g is strict in all its arguments, then $dom(g(f_1, \ldots, f_n)) \subseteq \bigcap_{i=1}^{n} dom(f_i)$.*

3 Extent Analysis of Data Fields

We will describe two analyses: the first is an "input to output"-analysis that considers where outputs can possibly be defined given where the inputs are defined. The second is an "output to input"-analysis that given a request for outputs derives what parts of the inputs will eventually be needed. The first analysis is cast as a traditional forwards analysis in the spirit of Mycroft [15]. The second analysis is formulated as a backwards analysis, see e.g. [12, 18].

We will define our analyses over a simple language of terms $\lambda x.t$ where t is a term over a many-sorted signature. Any sort τ in the signature is then a simple type with an interpretation $[\![\tau]\!]$ that is a cpo. Especially, the signature holds a boolean type β with interpretation $[\![\beta]\!]$ as the flat cpo B of booleans. The terms t are defined the usual way, as constants or variables of some simple type, and as expressions $f(t_1, \ldots, t_n)$ where f is a function symbol of type $\tau_1 \times \cdots \times \tau_n \to \tau$, where each τ_i, $1 \leq i \leq n$, is a simple type and t_i has type τ_i, $1 \leq i \leq n$. f can be either a constant function symbol or a variable. If the latter holds, then f may be given by a (possibly) recursive definition $f = t_f$. Any constant function symbol of type $\tau_1 \times \cdots \times \tau_n \to \tau$ is interpreted as a function $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$. Moreover, it is supposed to be strict in all its arguments – the only exception being the ternary *if*-function defined by (1).

The terms under consideration will now be of the form $\lambda x.t$, where x has the simple type σ and t has the simple type τ. Operations on data fields such as elementwise application, get communication and where-restriction will thus be expressed through λ-abstraction.

The main idea in the input-output analysis is to represent every data field $[\![\sigma]\!] \to [\![\tau]\!]$ with a predicate over $[\![\sigma]\!]$ such that the predicate is always true in any point where the data field is possibly defined. The output-input analysis, on the other hand, assumes an expression to be evaluated, which will in turn generate a number of calls to different elements of different data fields. For each data field, the aim of the analysis is to derive a predicate that is true in any point where the data field is possibly called as a result of evaluating the expression.

The predicates we consider are *not* mappings to the flat cpo B, but rather to the cpo $\overline{B}$ with the same elements as B but the ordering $\bot \sqsubset \mathit{false} \sqsubset \mathit{true}$. This is since where-restriction uses test for *false* to "undefine", so we want *false* to be "less defined" than *true*. An anomaly of $\overline{B}$ is that negation is not monotone w.r.t. the ordering $\mathit{false} \sqsubset \mathit{true}$. However, it turns out that the way our analyses

are defined, fixpoint iteration w.r.t. abstract versions of functions will still form monotone chains (see Theorem 4).

4 Input-Output Analysis

The predicates $[\![\sigma]\!] \to \overline{B}$ form a lattice where the bottom element is $\lambda x.\mathit{false}$, the top element is $\lambda x.\mathit{true}$, and the ordering is the standard one obtained by extending $\overline{B}$ pointwise. (Then $b \sqsubseteq b'$ implies $b(x) \implies b'(x)$ for all $x \in [\![\sigma]\!]$ for which $b(x) \neq \bot$ and $b'(x) \neq \bot$.) $\lambda x.\mathit{true}$ represents "no knowledge", i.e. a function with abstract representation $\lambda x.\mathit{true}$ can possibly be defined everywhere. We now define the following abstractions of functions $[\![\sigma]\!] \to [\![\tau]\!]$ given by expressions as defined above. (The logical connectives below are applied elementwise.)

1. $(\lambda x.\bot)^\# = \lambda x.\mathit{false}$.
2. $(\lambda x.c)^\# = \lambda x.\mathit{true}$ whenever $c \neq \bot$.
3. $(\lambda x.x)^\# = \lambda x.\mathit{true}$.
4. $(\lambda x.y)^\# = \lambda x.y^\#$ for $y \neq x$.
5. $(\lambda x.f(t_1,\ldots,t_n))^\# = (\lambda x.t_1)^\# \wedge \cdots \wedge (\lambda x.t_n)^\#$, if f is a constant function symbol distinct from *if*.
6. $(\lambda x.f(t_1,\ldots,t_n))^\# = \lambda x.f^\#(t_1,\ldots,t_n)$ if f is a variable.
7. $(\lambda x.\mathit{if}(b,t_1,t_2))^\# = [b \wedge (\lambda x.t_1)^\#] \vee [(\neg b) \wedge (\lambda x.t_2)^\#]$.
8. If f is defined by $f = t_f$, then $f^\# = t_f^\#$.
9. If f is a free variable, then $f^\#$ is assumed given.

The abstraction of $\lambda x.x$ is an approximation: it does not take into account that $\lambda x.x(\bot) = \bot$. In most cases, we will however only be interested in tabulating data fields for arguments which are total elements, and then this approximation makes no difference.

The following two rules can be derived from the above:

- $(f(g))^\# = f^\#(g)$, when f is a variable (get communication).
- $(f \setminus b)^\# = f^\# \setminus b$.

Given a recursive function definition $f = t_f$, the abstract recursive definition $f^\# = t_f^\#$ can be iterated to a greatest fixpoint starting with $f_0^\# = \lambda x.\mathit{true}$. The iteration can be extended to mutually recursive definitions in the usual way.

When $[\![\sigma]\!]$ is infinite the lattice of predicates $[\![\sigma]\!] \to \overline{B}$ has infinite height, and there is no guarantee that a fixpoint iteration will terminate. However, also for infinitely high lattices there are cases where the fixpoint iteration will converge in a finite number of steps:

Proposition 2. *Let f be a function of type $\sigma \to \tau$. If, for some j, $f_j^\#(x)$ is true for only finitely many $x \in [\![\sigma]\!]$, then there is a k such that $f_k^\#$ is a fixpoint.*

Proof. Then there will only be finitely many predicates $b\colon [\![\sigma]\!] \to [\![\tau]\!]$ such that $b \sqsubseteq f_j^\#$. Since the sequence $\{f_i^\#\}_{i=0}^\infty$ is monotonically decreasing (theorem 4 below) it must converge in a finite number of steps.

In order to use Proposition 2 in practice, it is also necessary to have a finite *representation* of the predicates under consideration. Furthermore, it must be possible to decide the finiteness of a predicate from its representation, and also to decide from their representations whether two predicates are equal or not. In the next section we will see an example.

We will now formulate and prove a correctness theorem for the input-output analysis. Since termination is not guaranteed, the theorem must be formulated in terms of the successive iterates $f_i^\#$, $i \geq 0$. For simplicity we prove the theorem only for a single recursive definition; the extension to mutual recursion is straightforward. First, a simple lemma:

Lemma 3. *if* $(b, x, y) \neq \bot \iff b \neq \bot \wedge [(b \wedge x \neq \bot) \vee (\neg b \wedge y \neq \bot)]$.

Proof. Immediate from (1).

Theorem 4. (Correctness of input-output analysis): *If f of type $\sigma \to \tau$ is recursively defined by $f = t_f$ then it holds, for all $i \geq 0$, that:*

1. $f_{i+1}^\# \sqsubseteq f_i^\#$.
2. *For all* $x \in [\![\sigma]\!]$, $f(x) \neq \bot \implies f_i^\#(x)$.

Proof. See appendix A.

4.1 Linear Constraints

A kind of predicate over infinite cpo's that seems possible to handle in many cases and is of importance in practice is systems of linear inequalities over $\mathbb{Z}_\bot^n$. Any predicate over $\mathbb{Z}_\bot^n$ formed by the logical connectives and linear inequalities can be written on disjunctive normal form $L_1 \vee \cdots \vee L_n$, where each L_i is a conjunct $l_{i1} \wedge \cdots \wedge l_{im(i)}$ of linear inequalities l_{ij} (i.e. a system of linear inequalities). Finiteness of an L_i can be decided by integer linear programming methods: thus, finiteness of $L_1 \vee \cdots \vee L_n$ can be decided by solving n ILP problems. In order to decide equality of two predicates b, b' of the form above, we can use ILP methods to decide whether P and P', the subsets of $\mathbb{Z}^n$ defined by b and b' respectively, are equal [14].

Thus, proposition 2 can be applied when, for all i greater than some j, $f_i^\#$ is a predicate formed from linear inequalities. If A is an affine mapping and $f^\#$ is a predicate of this form, then $\lambda i.f^\#(A(i))$ is such a predicate too: *thus, we can always analyze data fields defined by affine recurrences in this manner.* If we, at some point, arrive at a finite predicate, then we know that the iteration will terminate and we also have a method to decide termination.

As remarked before, there are ILP methods that can handle inequalities with symbolic parameters. Since finiteness and equality both can be checked by solving a number of ILP problems, it may appear that we will also be able to handle fixpoint iteration over parameterized predicates with linear inequalities. This is, however, not necessarily true. The reason is that the symbolic solutions

may contain conditions on the parameters. In order to decide equality, we must therefore be able to decide whether two disjuncts of symbolic solutions with conditions are equal or not. In the example below we give a recursive definition with symbolic parameters that can be analyzed, but where a slight modification of the definition makes a straightforward symbolic analysis difficult.

4.2 An Example: Matrix Multiplication

Consider the following definition of the classical matrix multiplication algorithm. A and B are $n \times n$-matrices, i.e. they have the abstract values $A^\# = \lambda ik.(1 \leq i,k \leq n)$ and $B^\# = \lambda kj.(1 \leq k,j \leq n)$, respectively:

$$\begin{aligned} mul(A,B,n) &= \lambda ij.C(A,B,i,j,n) \\ C(A,B) &= \lambda ijk.if(k = 0, 0, C(A,B,i,j,k-1) + A(i,k) * B(k,j)) \end{aligned} \tag{2}$$

(Here and henceforth we use the notational convention that "index arguments" of type σ, for data fields of type $\sigma \to \tau$, are introduced as lambda-bound variables in the right-hand side of the data field definition, whereas arguments that should be considered parameters to a data field are given as arguments in the left-hand side.) The system of abstract recursive functions becomes

$$\begin{aligned} mul(A,B,n)^\# &= \lambda ij.C^\#(i,j,n) \\ C^\# &= \lambda ijk.((k = 0) \vee \\ &\qquad ((k \neq 0) \wedge C^\#(i,j,k-1) \wedge A^\#(i,k) \wedge B^\#(k,j))). \end{aligned}$$

(For notational convenience, we drop the arguments A, B to $C^\#$.) This is a recursive predicate definition with a symbolic parameter n. Let us now perform the fixpoint iteration. It suffices to iterate w.r.t. $C^\#$ only, with $A^\#$, $B^\#$ and n given by the omitted call to $mul^\#$. $C_0^\# = \lambda ijk.true$,

$$\begin{aligned} C_1^\# &= \lambda ijk.((k = 0) \vee ((k \neq 0) \wedge C_0^\#(i,j,k-1) \wedge A^\#(i,k) \wedge B^\#(k,j))) \\ &= \lambda ijk.((k = 0) \vee ((1 \leq i,k \leq n) \wedge (1 \leq k,j \leq n))), \end{aligned}$$

and

$$\begin{aligned} C_2^\# &= \lambda ijk.((k = 0) \vee ((k \neq 0) \wedge C_1^\#(i,j,k-1) \wedge A^\#(i,k) \wedge B^\#(k,j))) \\ &= \text{ (some manipulations)} \\ &= C_1^\#. \end{aligned}$$

Thus, $C^\#(i,j,k) = C_2^\#(i,j,k)$ and

$$\begin{aligned} mul(A,B,n)^\# &= \lambda ij.C^\#(i,j,n) \\ &= \lambda ij.((n = 0) \vee ((1 \leq i,n \leq n) \wedge (1 \leq n,j \leq n))) \\ &= \lambda ij.((n = 0) \vee ((1 \leq i \leq n) \wedge (1 \leq j \leq n))). \end{aligned}$$

Note how we get two cases: $n = 0$ with no restrictions on i and j, reflecting that this is a nonstrict case where A, B are not used at all ($mul(A, B, 0)$ is 0 everywhere), and $n > 0$ where the restrictions on i, j are given by the corresponding restrictions on A, B.

In the example above the termination of the iteration could be decided, since we were able to manipulate two successive iterates symbolically so they became syntactically equal. It is instructive to consider an example where this does not seem possible. Let us make a slight modification to the definition of C above, so it uses $C(i, j, k-2)$ rather than $C(i, j, k-1)$. Intuitively, the "index set" for C will now be a number of "slices" parallel with the k-axis for $k = n, k = n-2, \ldots, k = 0$ (the result will now be defined for even n's only). This non-convex set can be expressed with linear equalities and inequalities *only* if we allow the size of the formula to grow with n: it will then have the form $(k = n \wedge \ldots) \vee ((k = n-2) \wedge \ldots) \vee \ldots \vee (k = 0 \wedge \ldots)$. But then we will not be able to decide termination by comparing formulae, since these will grow for every new iteration. Note, however, that if n is *fixed*, then also this iteration will eventually converge. This indicates that program specialization can be helpful to improve the success of extent analysis. The convergence will, however, require $O(n)$ steps rather than two.

4.3 A Second Example: List Length Analysis

The input-output analysis can sometimes be used to predict the length of lists. Thus, it may have applications also for analyzing more conventional functional programs. Recall that a nonempty list $a::L$ of type $List\ \tau$ can be interpreted as a function from natural numbers to $[\![\tau]\!]$, defined on the interval $[0, length(L) - 1]$, viz:

- $[\![a::L]\!](0) = [\![a]\!]$.
- $[\![a::L]\!](i) = [\![L]\!](i-1)$, $1 \leq i < length(L)$.

Thus, the theory of data fields can be applied to lists. In particular, input-output analysis can be performed. We will have $L^{\#} = \lambda i.0 \leq i < length(L)$ for (finite) lists L: this particular form of predicate is fully represented by $length(L)$, which we can take as the abstract version of L rather than the predicate. For functions over lists using cons, head, tail, empty list, test for empty list and conditional we can use the following abstract interpretation of lists:

- $NIL^{\#} = 0$.
- $(a::L)^{\#} = 1 + L^{\#}$.
- $tl(L)^{\#} = L^{\#} - 1$, when L is nonempty.
- $(L = NIL)^{\#} = (L^{\#} = 0)$.

Furthermore, we redefine the interpretation of the *if*-function to be

$$if(b, L_1, L_2)^{\#} = if(b, L_1^{\#}, L_2^{\#}).$$

For instance, we can analyze the following function that takes a list and puts an extra copy of each element in the resulting list:

$$double(L) = if(L = NIL, NIL, hd(L)::(hd(L)::double(tl(L)))).$$

The abstract version becomes

$$double^{\#}(L^{\#}) = if(L^{\#} = 0, 0, 1 + (1 + double^{\#}(L^{\#} - 1))).$$

This equation is not suitable to solve with fixpoint iteration. However, we can assume that $double^{\#}(L^{\#}) = a \cdot L^{\#} + b$ when $L^{\#} > 0$ and try to determine a and b. We obtain the following system of equations, where the latter comes from the "boundary condition" $double^{\#}(1) = 2 + double^{\#}(0)$, and $double^{\#}(0) = 0$:

$$\begin{aligned} a \cdot L^{\#} + b &= 2 + a \cdot (L^{\#} - 1) + b, \\ a + b &= 2 + b. \end{aligned}$$

This system has the solutions $a = 2$, b arbitrary. The *least* solution, subject to the constraint $L^{\#} \geq 0 \implies double^{\#}(L^{\#}) \geq 0$, is $a = 2$, $b = 0$. By theorem 4, we can then safely assume that $length(double(L)) = 2 \cdot length(L)$.

5 Two-Step Termination of Input-Output Analysis

In the first example in Sect. 4.2, the fixpoint iteration terminated in two steps. A question thus arises: in which cases do this occur? Below we will give a class of recursive functions where the fixpoint iteration will *always* terminate in two steps. Consider the following form of recursive definition:

$$y = \lambda x.if(p(x), g(x), f(h(x), y(b(x))))$$

where f is strict in both arguments, and p, g and h are not dependent of y. We have

$$y^{\#} = \lambda x.[(p(x) \wedge g^{\#}(x)) \vee ((\neg p(x)) \wedge h^{\#}(x) \wedge y^{\#}(b(x)))].$$

If we define $P(x) = p(x) \wedge h^{\#}(x)$ and $A(x) = (\neg p(x)) \wedge h^{\#}(x)$, then we can write

$$y^{\#} = \lambda x.[P(x) \vee (A(x) \wedge y^{\#}(b(x)))].$$

Let $\mathcal{P}$ and $\mathcal{A}$ denote the sets defined by P and A, respectively. The following hypothesis is made on the recursive definition:

H1 All the points in $\mathcal{A}$ are "computable points", i.e.,

$$\forall x \in \mathcal{A}, (b(x) \in \mathcal{P}) \vee (b(x) \in \mathcal{A}),$$

or, equivalently,

$$A(x) \implies P(b(x)) \vee A(b(x)) \quad \text{for all } x.$$

Notice that **H1** is equivalent to the existence of a "used-def" chain between any point of $\mathcal{A}$ and some point in $\mathcal{P}$ (or, possibly, a cyclic dependence within $\mathcal{A}$). This is reminiscent to dependence analysis in automatic parallelizers.

Proposition 5. ***H1*** *implies that the fixpoint iteration for* $y^\#$ *terminates in two steps.*

Proof. Assume that **H1** holds. We have $y_0^\# = \lambda x.true$, and subsequently $y_1^\# = \lambda x.P(x) \vee A(x)$. Furthermore,

$$y_2^\# = \lambda x.P(x) \vee [A(x) \wedge (P(b(x)) \vee A(b(x)))].$$

We have $y_1^\# = y_2^\#$ iff $y_1^\# \sqsubseteq y_2^\#$ and $y_2^\# \sqsubseteq y_1^\#$. The latter is always true. $y_1^\# \sqsubseteq y_2^\#$ holds if, for all x,

$$A(x) \implies A(x) \wedge [P(b(x)) \vee A(b(x))],$$

that is:

$$A(x) \implies P(b(x)) \vee A(b(x))$$

which is exactly **H1**.

We can even give the solution as a closed formula:

$$y^\# = \lambda x.[(p(x) \wedge g^\#(x)) \vee ((\neg p(x)) \wedge h^\#(x))].$$

In the example in Sect. 4.2, the recursive definition of C obeys **H1** with $p = \lambda ijk.(k = 0)$, $g^\# = \lambda ijk.true$, $h^\# = \lambda ijk.1 \leq i, j, k \leq n$ and $b = \lambda ijk.(i, j, k-1)$. If we, however, change b to $\lambda ijk.(i, j, k-2)$ as in the modified example, then **H1** does not hold: some calls from $\mathcal{A}$ will "fall off the edge" and cause a successive "undefinition" of interior points of $\mathcal{A}$ as the iteration proceeds.

6 Output-Input Analysis

Let us now turn to the opposite problem: given a *request* for a part of a data field, what parts of the inputs will eventually be requested? For a recursively defined function we also want to know: which recursive calls will be made? The result of the analysis will be one predicate for each input (recursive function), that is *true* in the requested points, or call arguments, for that input (recursive function). The basic idea is very simple: just perform a symbolic execution, and for each call (request for input), add the argument tuple to the predicate (i.e. OR it).

The framework is that of backwards analysis. Backwards analysis can be thought of as an *environment* being propagated from a function call to its arguments. Formally, the environments can be defined as *projections* [18]: a projection over a cpo C is a continuous function $p: C \to C$ such that:

- $p \sqsubseteq ID$ (ID is the identity function).
- $p \circ p = p$. (idempotence)

Note that for every predicate $b: C \to \overline{B}$, $\lambda f.(f \setminus b)$ defines a projection over each function cpo $C \to C'$. So a predicate can be seen as an environment telling in which points a function is to be called.

The analysis is presented in the spirit of Hughes [12]. Propagation of a constraint b through an expression t to a (data field) variable f is denoted by $b -t \to f$: this is a predicate that tells which points of f that can be evaluated, given that the expression t is evaluated in the points defined by b.

For all data field variables f of type $\sigma \to \tau$ and recursively defined data field variables g of type $\sigma' \to \tau'$, given by $g = t_g$, we define

$$g_f^{\#}(b) = b -t_g \to f.$$

$g_f^{\#}$ transforms the predicate $b: [\![\sigma']\!] \to \overline{B}$ into a predicate $[\![\sigma]\!] \to \overline{B}$. If g is not recursively defined (say, a parameter), then we define

$$g_f^{\#}(b) = \lambda x.\mathit{false}.$$

When f is given, the idea is to find recursive definitions for all $g_f^{\#}(b)$ by unfolding $b -\lambda x.t_g \to f$ according to the rules defining $b -t \to f$. The resulting recursive system of definitions can then in principle be solved by a fixpoint iteration over the lattice $([\![\sigma']\!] \to \overline{B}) \to ([\![\sigma]\!] \to \overline{B})$. Contrary to input-output analysis, the fixpoint iteration for output-input analysis starts with the least element $\lambda b.\lambda x.\mathit{false}$ in $([\![\sigma']\!] \to \overline{B}) \to ([\![\sigma]\!] \to \overline{B})$. Thus, the solution, if found, will be a least fixpoint.

Assume that we know $g_f^{\#}$. For a call $g \setminus b$, $g_f^{\#}(b)$ can then be evaluated to yield a predicate that is *true* in those points where f becomes requested due to this call. We now present the rules defining $b -t \to f$:

- $b -\lambda x.c \to f = \lambda x.\mathit{false}$ for any constant c, including $\bot$.
- $b -\lambda x.y \to f = \lambda x.\mathit{false}$ for any (simple type) variable y, including x.
- $b -\lambda x.g(t_1, \ldots, t_n) \to f = b -\lambda x.t_1 \to f \vee \cdots \vee b -\lambda x.t_n \to f$, if g is a constant function symbol distinct from *if*.
- $b -\lambda x.\mathit{if}(v, t_1, t_2) \to f = [b -\lambda x.v \to f] \vee [(b \wedge \lambda x.v) -\lambda x.t_1 \to f] \vee [(b \wedge \lambda x.\neg v) -\lambda x.t_2 \to f]$.
- $b -\lambda x.f(t_1, \ldots, t_n) \to f = b' \vee f_f^{\#}(b')$, where $b' = \lambda x'.\exists x.(x' = (t_1, \ldots, t_n) \wedge b(x))$.
- $b -\lambda x.h(t_1, \ldots, t_n) \to f = h_f^{\#}(b')$ when h is a higher order variable distinct from f, with b' as above.

The following is an informal motivation for the analysis. First order constants and variables cannot generate any calls to f, thus $\lambda x.\mathit{false}$ is returned for them. For strict constant function symbols all arguments must be evaluated: therefore all possible calls to f, resulting from the evaluation of any argument, are collected. For the *if*-function, the first disjunct records any calls to f through the evaluation of v. The second and third disjuncts, respectively, record the calls to f that may result from evaluating t_1 and t_2 under the conditions v and $\neg v$, respectively.

The rules for propagation through calls to data fields deserve special mentioning. In the rule for $b\ {-}\lambda x.f(t_1,\ldots,t_n) \rightarrow f$ the first disjunct records the call, with the possible values of the arguments given by b. The existentially quantified x represents the "old" possible values of arguments; $x' = (t_1,\ldots,t_n)$ gives the new binding of argument (note that x in general will occur in $(t_1,\ldots,t_n)$, thus x' is defined in terms of x). $b(x)$ essentially means that b decides the possible values of the old arguments. The second disjunct gives the recursive call to $f_f^{\#}$.

The rule for $b\ {-}\lambda x.h(t_1,\ldots,t_n) \rightarrow f$ is similar: the call itself is not recorded but the recursive call to $h_f^{\#}$ must still be made, to cater for any possible calls to $f_f^{\#}$ resulting from the call.

The following rule for where-restriction can be derived:

$$b\ {-}(\lambda x.t) \setminus v \rightarrow f = (b\ {-}v \rightarrow f) \vee ((b \wedge v)\ {-}\lambda x.t \rightarrow f).$$

Some natural properties of $b\ {-}t \rightarrow f$ can be deduced from the above. First, for any terms t without recursively defined variables, $b\ {-}t \rightarrow f = \lambda x.\mathit{false}$ for any b, f. This also means that for conditions v in conditionals or where-restrictions not depending on data fields, the rules for *if* and where-restriction simplify to

- $b\ {-}\lambda x.\mathit{if}(v,t_1,t_2) \rightarrow f = ((b \wedge v)\ {-}\lambda x.t_1 \rightarrow f)(\vee(b \wedge \neg v)\ {-}\lambda x.t_2 \rightarrow f)$ and
- $b\ {-}(\lambda x.t) \setminus v \rightarrow f = (b \wedge v)\ {-}\lambda x.t \rightarrow f$,

respectively. Second, it is easy to see that $\lambda x.\mathit{false}\ {-}\lambda x.t \rightarrow f = \lambda x.\mathit{false}$ for all t, f. This rule is important for achieving termination when evaluating abstract functions.

7 Related Work

The analyses here are formulated in a purely functional setting, but the main applications of the techniques seem to be found for array-like recursive definitions. Related work thus exists for both functional languages and imperative languages. The formulation of the input-output analysis is influenced by Mycroft's thesis [15], and the output-input analysis is inspired by the work by Wadler and Hughes on projections for backwards analysis [18]. Data fields over integer tuples restricted by linear inequalities are akin to array comprehensions in Haskell, with the exception that an array comprehension must have its extent specified explicitly (and thus it need not be derived by extent analysis). Anderson and Hudak [1] apply dependence tests to array comprehensions in order to schedule them for thunkless evaluation: the same applies to the corresponding class of data fields if the extent analysis succeeds. Optimizing affine space-time mappings should also often be possible to apply in this case, see, for instance, [13].

Some analyses on graph based intermediate forms for array-based eager functional languages can be seen more or less as special cases of our input-output analysis. In particular, this applies to the *Size Inference* for vectors in VCODE

[4] and the *Build-in-Place Analysis* of the SISAL compiler described by Feo, Cann and Oldehoeft [8].

For imperative programs, we would like to mention the array reference analysis by fixed point iteration by Duesterwald, Gupta and Soffa [6]. Optimization of array bounds checking [9] is also related to our work: in fact, our analyses could be used for this purpose. Finally, it is interesting to compare the output-input analysis with Dijkstra's *predicate transformers* for imperative programs [5].

8 Conclusion and Future Work

We have presented two methods to perform *extent analysis* on recursively defined data fields. *Input-output analysis* finds where a data field is (possibly) defined given that we know where the inputs are defined. This is a forwards analysis. *Output-input analysis* derives the (possibly) required parts of inputs from requested outputs. It is a backwards analysis that propagates environments "inwards". For the input-output analysis, the abstract domain consists of predicates defined over the same cpo as the data field to be analyzed. If that cpo is not finite, then the abstract domain will not be finite either and fixpoint iteration will in general not terminate. For linear inequalities over tuples of integers the situation is different, though: here, symbolic methods can be used to decide equality between predicates. Also, finiteness can be decided, and if an iterate is finite then the fixpoint iteration will eventually terminate. Linear inequalities constitute an important special case with applications to programs with arrays. We proved a weak form of correctness for the input-output analysis: if it terminates, then the resulting predicate is guaranteed to be true in all points where the analyzed data field is distinct from $\perp$. Furthermore, we presented a special case of recursive definition, where the fixpoint iteration for input-output analysis under certain conditions is guaranteed to terminate in two steps (and then the solution has a closed form, so no iteration is actually needed). It thus seems plausible that the input-output analysis can be automated for at least some interesting instances of recursively defined data fields.

The situation for the output-input analysis is more problematic, however. Since the abstractions are predicate transformers rather than predicates, it seems less likely that there will be interesting cases that can be handled symbolically. An example which displays some of the problems is developed in [14]. Even if the cpo's under consideration are finite, the higher order function lattice for the abstract values will be prohibitively large for all but trivial cases. Yet, it seems of interest to continue pursuing analysis methods for "compile-time demand propagation", especially for languages with lazy semantics.

The present analyses are cast in the framework of abstract interpretation, with the solution of fixpoint equations as the primary method to find the solution. One can wonder whether suitable type systems can be designed, so type checking algorithms can be employed instead.

9 Acknowledgements

We would like to thank the anonymous referees for their valuable comments.

References

1. S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 137–149. ACM, June 1990.
2. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.
3. G. E. Blelloch and G. W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.
4. S. Chatterjee, G. E. Blelloch, and A. L. Fisher. Size and access inference for data-parallel programs. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 130–144, June 1991.
5. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.
6. E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *Proc. ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.
7. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.
8. J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.
9. R. Gupta. A fresh look at optimizing array bound checking. In *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.
10. 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.
11. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, Dec. 1986.
12. J. Hughes. Compile-time analysis of functional programs. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, chapter 5, pages 117–153. Addison-Wesley, Reading, MA, 1989.
13. B. Lisper. Linear programming methods for minimizing execution time of indexed computations. In *Proc. Int. Workshop on Compilers for Parallel Computers*, pages 131–142, Dec. 1990.
14. B. Lisper and J.-F. Collard. Extent analysis of data fields. Research report, LIP, ENS Lyon, France, 1994. ftp: `lip.ens-lyon.fr`.
15. A. Mycroft. *Abstract interpretation and optimizing transformations for applicative programs.* PhD thesis, Computer Science Dept,. Univ. of Edinburgh, 1981.
16. D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, pages 38–50, December 1990.

17. G. L. Steele Jr. and W. D. Hillis. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. Technical Report PL86–2, Thinking Machine Corporation, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts 02142, May 1986.
18. P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Proc. Functional Programming Lang. and Computer Arch.*, pages 385–407, Berlin, Sept. 1987. Volume 274 of *Lecture Notes in Comput. Sci.*, Springer-Verlag.
19. 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 Proof of Theorem 4

1. The only nonmonotone operation that may occur in $t_f^\#$ is negation "$\neg$". $f_{i+1}^\# \not\sqsubseteq f_i^\#$ can thus happen only if $f^\#$ occurs below a negation in $t_f^\#$, and this may happen only if f occurs in a condition b in some subterm $\mathit{if}(b, t_1, t_2)$ of t_f. However, the subterms resulting in $t_f^\#$ from b are b and $\neg b$. These terms will contain f rather than $f^\#$, and thus $f^\#$ will not occur below possible negations. (Note that for boolean expressions that do *not* occur as conditions to *if*, negation is treated as a strict unary constant function symbol.)

2. By induction on i:

- $i = 0$: vacuously true.
- $i > 0$: assume true for $i-1$. $f_i^\#$ equals $t_f^\#[f_{i-1}^\#/f^\#]$ ($t_f^\#$ with $f_{i-1}^\#$ substituted for $f^\#$). Proof by structural induction over the first order terms t of type τ forming the possible terms $\lambda x.t = t_f^\#[f_{i-1}^\#/f^\#]$ under consideration:
  - $t = \bot$, $t = c$, $t = x$: directly true.
  - $t = y \neq x$: true by definition, since $y^\#$ is assumed given.
  - $t = g(t_1, \ldots, t_n)$, where g is a constant function symbol $\neq \mathit{if}$: assume true for $t_1, \ldots, t_n$. For any $y \in [\![\sigma]\!]$ we obtain, using proposition 1, that $\lambda x.t(y) \neq \bot \implies \lambda x.g(t_1, \ldots, t_n)(y) \neq \bot \implies \lambda x.t_1(y) \neq \bot \wedge \cdots \wedge \lambda x.t_n(y) \neq \bot \implies (\lambda x.t_1)^\#(y) \wedge \cdots \wedge (\lambda x.t_n)^\#(y) = (\lambda x.t)^\#(y)$.
  - $t = f(t_1, \ldots, t_n)$: Then $(\lambda x.t)^\# = f_{i-1}^\#(t_1, \ldots, t_n)$. By the induction on i, the property holds.
  - $t = h(t_1, \ldots, t_n)$, where h is a free variable: true by definition, since $h^\#$ is assumed given.
  - $t = \mathit{if}(b, t_1, t_2)$: Assume, for any y, that $\lambda x.\mathit{if}(b, t_1, t_2)(y) \neq \bot$. Then, by lemma 3 and induction on t_1 and t_2, $\mathit{if}(\lambda x.b(y), \lambda x.t_1(y), \lambda x.t_2(y)) \neq \bot \iff \lambda x.b(y) \neq \bot \wedge [(\lambda x.b(y) \wedge \lambda x.t_1(y) \neq \bot) \vee (\neg \lambda x.b(y) \wedge \lambda x.t_2(y) \neq \bot)] \implies (\lambda x.b(y) \wedge (\lambda x.t_1)^\#(y)) \vee ((\neg \lambda x.b(y)) \wedge (\lambda x.t_2)^\#(y)) \implies [(\lambda x.b \wedge (\lambda x.t_1)^\#) \vee ((\neg \lambda x.b) \wedge (\lambda x.t_2)^\#)](y) \iff (\lambda x.\mathit{if}(b, t_1, t_2))^\#(y)$.

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