Data Flow Analysis of Recursive Structures

Albert Cohen[1], Jean-François Collard[1,2], Martin Griebl[3]

[1] PRISM, Univ. de Versailles, 45 Av des Etats-Unis, 78035 Versailles, FRANCE
[2] Kungliga Tekniska Högskolan, Electrum 204, 164 40 Kista, SWEDEN
[3] FMI, Universität Passau, Innstraße 33, 94032 Passau, GERMANY

Abstract. Most imperative languages only offer arrays as "first-class" data structures. Other data structures, especially recursive data structures such as trees, have to be manipulated through explicitly management of memory. On the other hand, recursion in the flow of control also is an open problem in automatic parallelization. To help and solve this problem, this paper proposes a data flow analysis for both data and control recursive structures.

Keywords: Static analysis, automatic parallelization, compiler

1 Introduction

Data structures in imperative languages are in general of little variety. In most cases, such structures are arrays, records and unions of basic types. The advantage of such restrictions is that the the compiler can construct a finite representation of data structures. When possibly infinite data structures are to be used, most imperative languages rely on pointers and dynamic memory allocation. For instance, a binary tree can be expressed and manipulated using a record of one datum and two pointers. This record has finite size, easing the work of the compiler (and of the compiler writer). Of course, there are languages which offer recursive data structures without explicit manipulation of memory. Most logic or functional languages do so. As an example of imperative language, see for instance CLU [1]. However, we believe most languages do not provide this feature because of the lack of suitable compile-time analyses. The aim of this paper is to report preliminary results on the data flow analysis of imperative languages offering both recursive control and recursive data structures. Section 2 defines our program model and introduce a small toy language. Section 3 describes our data flow analysis. Section 4 gives two extended examples.

2 The Source Language

We present below a subset of LEGS [2], a toy language that allows to cleanly define data and control recursive structures. It is enough here to say that LEGS embodies restrictions and assumptions we have had to make on the way recursive structures are defined and handled. (See Section3.1.)

2.1 Defining Recursive Data Structures

Let E by a finite alphabet and ϵ the empty word. To each word in the language corresponds an element of the data structure. In this paper, we only consider two classes of languages on E: those where concatenation is the usual one, and those where concatenation is commutative. (We will say that the data structure itself is commutative or not.) For instance, if $E = \{l, r\}$, then a non commutative concatenation labels the nodes of a binary tree. Otherwise, if $l.r = r.l$, then an "array" is described.

2.2 Recursive Control Structure

Recursive control is expressed by (possibly nested) `explore` constructs, each consisting of three parts: labels to recursive calls, a predicate $\mathcal{P}$ controlling the recursion, and a body. Statements in the body are of two kinds: recursive calls and "regular" statements (assignments, I/O, etc.). Statements of both kind may be arbitrarily mixed and are separated by a semicolon. All statements are labeled. Recursive calls are done by key word `recurse`.

The main benefit of this construction, when compared to guarded recursive calls, is that the execution of the body occurs *if and only if* the predicate of the `explore` holds. (Consider a simple recursive function, say $f(x) = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-2)$. Deriving that f is defined for even integers already needs some semantical analysis: an issue we wanted to avoid here.)

Definitions The set of assignment labels $\{S_1, \ldots, S_n\}$ is denoted by STMTS. The set of labels of recursive calls in all `explore` constructs is denoted by LABELS ($= \{C_1, \ldots, C_n\}$). We also define UP $= \{{C_1}^{-1}, \ldots, {C_n}^{-1}\}$, i.e., the set of inverses of recursive call labels. Any execution of an assignment can thus be uniquely labeled by a word in LABELS*.STMTS. Such a word is called a *control word*.

Since a statement is a static object (a "line" in the program), we call *operations* the (run-time) instances of statements. Statements are labeled in the program text. Labeling operations can be done using a pair $\langle S, w\rangle$, where w is the control word.

The textual order on labels is denoted by $\lhd$. For instance $\mathtt{b} \lhd S \lhd \mathtt{f}$ in Figure 2. The order in which statements are executed is thus simply given by the usual lexicographic order, denoted $\ll$, on LABELS*.STMTS. Let w, w' be two words labelling instances of Statements S and S', respectively. (So, $w \in \text{LABELS}^*.S$ and $w' \in \text{LABELS}^*.S'$[4].) Thus: $\langle S', w'\rangle \prec \langle S, w\rangle \;\equiv\; w' \ll w$. A formal definition of the order is:

$$
\begin{aligned}
w' \ll w \;\equiv\; & \exists u \in \text{LABELS}^*, \\
& ((\exists x \in \text{LABELS and } \exists v \in \text{LABELS}^*.S) \text{ or } (x = S \text{ and } v = \epsilon)), \\
& ((\exists x' \in \text{LABELS and } \exists v' \in \text{LABELS}^*.S') \text{ or } (x' = S' \text{ and } v' = \epsilon)), \\
& w = u.x.v, \; w' = u.x'.v', \text{ and } x' \lhd x
\end{aligned} \tag{1}
$$

2.3 Accessing Elements of Data Structures

To access an element of a data structure, the name of the structure is followed, between brackets, by a word in the language of the data structure: *structure* `[` *access word* `]`. *access word* has to be the image of the current control word, say w, by a substitution σ. This image is denoted by $w\sigma$. A substitution has the intuitive usual syntax, i.e. $\{x/a, y/ab\}$ is written as `{x/a,y/ab}`. *Notice that letters which do not appear in the*

[4] Note that there is some redundancy in this notation since the control word of S includes exactly one S, at the very last position. The definition of order is still valid when $S = S'$.

substitution are implicitly replaced by ϵ, since the letters in the "subscripting" word have to belong to the language describing the data structure. Figures 1 and 2 give two examples whose data flow analyses will be derived in following sections. The former is an exploration of a binary tree. The latter is a recursive program over an array-like data structure (and actually an excerpt from a program to recursive sort an array).

```
explore L while P(#L) {
  explore R while #R < q {
    S: tree[{L/l}.{R/r}] := tree[{L/l}.l^-1.{R/r}];
    R: recurse; }
  L: recurse; }
```

Fig. 1. Example LEGS program. `tree` is a binary tree-shaped data structure described by the language on $\{\mathtt{l}, \mathtt{r}\}$. `P` is some affine predicate.

```
explore  f, b  while  #f + #b < N {
  b : recurse ;
  S: A[{ f/u, b/u^(-1)}] :=
     A[{ f/u, b/u^(-1)} . u^(-1)];
  f : recurse ;
}
```

```
int A[ K ] ;
F(g,p) begin
    if(p<N) then F(g-1,p+1) ;
    A[g] := A[g-1] ;
    if(p<N) then F(g+1,p+1)
end
```

`A` is a one-dimensional, possibly infinite, data structure, defined on alphabet $\{\mathtt{u}\}$ (so, it is a list). The LEGS program above on the left is equivalent to the pseudo-Pascal program on the right (where the initial call is `F(0,0)`), the difference being that parameter K has to be given a value in Pascal.

Fig. 2. Example LEGS program.

3 Data Flow Analysis

The purpose of data flow analysis is to find, for any read in a given operation, which previous write produced the read value.

Let S (S') be a statement reading (writing into) data structure `A` using substitution σ (σ'), and whose control word is w (w'). The fact that memory cells accessed by $\langle S, w\rangle$ and $\langle S', w'\rangle$ are the same is denoted by $\mathtt{A[}\ w\sigma\ \mathtt{]}{:=:}\mathtt{A[}\ w'\sigma'\ \mathtt{]}$.

So, for any instance $\langle S, w\rangle$ of Statement S, we are looking for the last instance w' of S' that wrote into $\mathtt{A[}\ w\sigma\ \mathtt{]}$, i.e, such that:

$$\mathtt{A[}\ w\sigma\ \mathtt{]}{:=:}\mathtt{A[}\ w'\sigma'\ \mathtt{]} \;\Leftrightarrow\; w\sigma = w'\sigma' \tag{2}$$

$\langle S', w'\rangle$ is then called the *source* of the read in $\langle S, w\rangle$.

Our framework for data flow analysis, first developed by Feautrier[3], consists in two steps. For a given statement S:

- Construct the set of instances of S' that are possible sources of $\langle S, w\rangle$. This set $\mathbf{Q}_{S'S}(w)$[5] is built from all operations $\langle S', w'\rangle$ such that: (1) $\langle S', w'\rangle$ is actually executed (2) $\langle S', w'\rangle$ writes into the memory cell that $\langle S, w\rangle$ reads, and (3) $\langle S', w'\rangle$ executes before $\langle S, w\rangle$, written $\langle S', w'\rangle \prec \langle S, w\rangle$. I.e., $\mathbf{Q}_{S'S}(w) = \{w' \mid \mathcal{P}(w'), (2), w' \ll w\}$.
- The source is $K_{S'S}(w) = max_{\ll}(\mathbf{Q}_{S'S}(w))$.

It may be the case that the source does not exist. $K_{S'S}(w)$ is then by definition equal to the *undefined operation* $\perp$. By convention, $\perp$ is the earliest operation in the program, i.e.,

$$\forall S, \forall w,\ \perp \prec \langle S, w\rangle \quad \vee \quad \langle S, w\rangle = \perp. \tag{3}$$

For technical reasons, we cannot handle directly the disjunction in the definition (1) of $w' \ll w$. So, we "split" $\mathbf{Q}_{S'S}(w)$ into n *subsets of possible sources* $\mathbf{Q}^1_{S'S}(w), \ldots, \mathbf{Q}^n_{S'S}(w)$ according to each disjunct. We solve them in turn, the results being $K^1_{S'S}(w)$,..,$K^n_{S'S}(w)$. Each $K^i_{S'S}(w)$, $1 \leq i \leq n$, is a nested conditional whose predicates depend on w and whose leaves include the maximum element according to order $\prec$. Notice that this order is strict.

We then combine the intermediate results in any order, using rules similar to those given in [3]: two intermediate results are merged by plugging one of them at the other's leaves. Leaves are equal to the lexicographical maximum of the two corresponding leaves. Leaves governed by contradictory predicates are dismissed. Obviously, this phase has to be repeated for all possible statements S', and the intermediate results have to be combined.

3.1 Program Model: Restrictions on Input Programs

Let $\#_l(w)$ denote the number of occurrences of letter l in word w. In this preliminary study of data flow analyses for regular structures, we assume the following restriction on the input program:

Recursion Predicate Let w be a control word on $\{C_1, \ldots, C_n\}$. Then, we restrict ourselves to predicates $\mathcal{P}$ of `explore` that are conjunctions of affine (in)equalities in the number of occurrences of labels of LABELS in control words. Notice that predicate $\mathcal{P}$ does not have to be a function of recursion depth.

Substitutions Substitutions are restricted to mappings from one letter (of the language of control words) to a word (in the language of the data structure).

The Word Problem Finding the elements in $\mathbf{Q}_{S'S}(w)$ requires to check that `A[` $w\sigma$ `]:=:A[` $w'\sigma'$ `]`, i.e. that $w\sigma = w'\sigma'$. Checking that two words are equal, known as "the word problem", is undecidable. We thus restricted ourselves to words for which a normal form can be defined.

[5] When clear from the context, both S and S' will be dropped.

3.2 Numbered Occurrence Languages

The *pumping* $[e_1^{i_1}, \ldots, e_d^{i_d}]$ of an alphabet $\Sigma = \{e_1, \ldots, e_d\}$ by a mapping of Σ into Z which maps e_j into i_j is the language built of all words containing exactly i_j occurrences of e_j, for all j. An *affine parametrized pumpings*, denoted by

$$\frac{[e_1^{i_1}, \ldots, e_d^{i_d}]}{A(i_1, \ldots, i_d)}$$

, is the union of all pumpings such that parameters $i_1, \ldots, i_d$ satisfy the system A of affine (in)equalities. The *inverse of a pumping* P has no sense by itself but we define the notation $w.P^{-1}$, where w is a word as the set of prefixes of w which corresponding suffixes are elements of P. P^{-1} can be seen either as a "selective" back space over w, or as a back space "recording" what has been erased. is the language $\{w' \mid \exists w \in P, w' = w^{-1}\}$.

A pumping is finite if the corresponding language is finite. This can be checked statically by checking that upper bounds on all parameters exist.

More generally, a *numbered occurrence language*, NOL, is given by p words $M_1..M_p$ (possibly empty), $p-1$ pumpings (or pumping inverses) $P_1..P_{p-1}$, and a conjunction A of affine (in)equalities. An NOL is denoted by

$$L = \frac{M_1.P_1. \cdots .M_{p-1}.P_{p-1}.M_p}{A}.$$

(Notice that several pumpings in an NOL may share integer parameters that are constrained by A.) A *simple NOL* has no pumping inverses.

Lemma 1. *If predicate $\mathcal{P}$ controlling the* **`explore`** *is affine (cf Section3.1) and if the data structure is commutative, then any subset of possible sources $\mathbf{Q}^j_{S'S}$ (for any j) is a NOL.*

Proof. Let $w\sigma$ and $w'\sigma'$ be the two words defined in (2). Since the structure is commutative, both words have a normal form, which are $w\sigma = e_1^{\#_{e_1}(w\sigma)} \ldots e_d^{\#_{e_d}(w\sigma)}$ and $w'\sigma' = e_1^{\#_{e_1}(w'\sigma')} \ldots e_d^{\#_{e_d}(w'\sigma')}$, respectively. These normal forms are equal iff:

$$\bigwedge_{i=1}^{d} \#_{e_i}(w\sigma) = \#_{e_i}(w'\sigma') \tag{4}$$

Thus, due to (1), we have $\mathbf{Q}^j_{S'S} = \frac{w.v^{-1}.x^{-1}.x'.v'}{\mathcal{P}(w') \ \wedge \ (4)}$, hence the lemma.

In the case of non commutative structures, sets of possible sources will be *approximated* using NOLs. This clearly implies that our data flow analysis may yield imprecise (but always correct) results.

Computing Lexicographical Maxima on Simple NOLs Given a word M and a pumping P on alphabet $\{C_1, ..C_n\}$, computing the lexicographical maximum of the words $M.P$ is done as follows:

- Change of variables: let $x_1, ..x_n$ be the parameters of letter of the alphabet in P. Each x_i may be an (affine) function of the original parameters. (These new variables are ordered in the same order as letters of the alphabet.)
- Compute the lexicographical maximum $(u_n, u_{n-1}, ...u_1)$ of $(x_n, x_{n-1}, ...x_1)$, under the (affine) constraints A. This is exactly what a software such as PIP[4] was crafted for.
- The lexicographical maximum in $M.P$ is $C_n^{u_n}.C_{n-1}^{u_{n-1}}....C_1^{u_1}$

Example: What is the lexicographical maximum of $\frac{[a^{n+i-j}, b^j]}{0 \leq i \leq n, 0 \leq j \leq n}$? $x_1 = n + i - j, x_2 = j \Rightarrow lexmax(x_2, x_1) = (n, n) \Rightarrow lexmax(M.P) = b^n.a^n$.

Lexicographical Order in the Presence of Pumping Inverses We believe that deciding the order of two words $w.P^{-1}, w'.P'^{-1}$ is impossible in general, because the answer depends on the actual values of w and w', even when $w = w'$. However, some restricted results happened to be sufficient in practice for most (simple) programs. In the case of binary recursion, we have the following lemma:

Lemma 2. *If $C_1 \lhd S \lhd C_2$ (i.e., the recursion is inorder) then, for all w, $\phi > \phi'$ and $\beta \geq \beta'$:*

$$w.S^{-1}.[C_1^{\phi}, C_2^{\beta}]^{-1}.C_1^{-1}.S \ll w.S^{-1}.[C_1^{\phi'}, C_2^{\beta'}]^{-1}.C_1^{-1}.S \tag{5}$$

Proof. For both words to be defined, w must be such that

$$w = x.C_1.[C_1^{\phi-\phi'}, C_2^{\beta-\beta'}].[C_1^{\phi'}, C_2^{\beta'}].S,$$

which is equal to $x.C_1.[C_1^{\phi-\phi'-1}, C_2^{\beta-\beta'}].C_1.[C_1^{\phi'}, C_2^{\beta'}].S$. Then $x.S$ is the left-hand side expression in (5), and $x.C_1.[C_1^{\phi-\phi'-1}, C_2^{\beta-\beta'}].S$ the right-hand side. The first letters after the common prefix are S and C_1, respectively. Moreover, $S \lhd C_1$, hence the lemma.

Similar lemmas for binary preorder and postorder recursion can easily be found.

Hard and Soft Bottoms As we said, $\mathbf{Q}_{S'S}(w)$ is split into subsets $\mathbf{Q}^1_{S'S}, \ldots, \mathbf{Q}^4_{S'S}$ in the course of the computation. Each $\mathbf{Q}^j_{S'S}$ may be empty for two reasons:

- $\mathcal{P}(w')$ and (2) cannot (perhaps simultaneously) be satisfied. Then, *all others* $\mathbf{Q}^i_{S'S}$, $i \neq j$, are empty too. We then know that the source of $\langle S, w\rangle$ cannot be an instance of S', and all the $K^j_{S'S}$ can be set to a *hard bottom* value, denoted by $\underline{\perp}$.
- The current disjunct of (1) is not compatible with $\mathcal{P}(w')$ and/or (2). Even though $\mathbf{Q}^j_{S'S}$ is then empty, other subsets $\mathbf{Q}^i_{S'S}$ may not. So, the maximal element $K^j_{S'S}$ of $\mathbf{Q}^j_{S'S}$ should be set to a *soft bottom* value, denoted by $\perp$.

Notice that distinguishing hard bottoms from soft bottoms is not semantical, but speeds up computations since this allows to prune impossible cases in other conditionals.

4 Examples

4.1 First Example

The set $\mathbf{Q}(w)$ of all previous instances w' of S that wrote in the memory cell read by w is:

$$\begin{aligned}\mathbf{Q}(w) = \{w' \mid P(\#_L(w'))\ , \#_R(w') < q, & \quad (6)\\ w'\{L/l\}.\{R/r\} = w\{L/l\}.l^{-1}.\{R/r\}, & \quad (7)\\ w' \ll w\} & \quad (8)\end{aligned}$$

(8) boils down to:

$$w' \ll w \quad \Leftrightarrow \quad \#_L(w') < \#_L(w) \ \vee \ (\#_L(w') = \#_L(w) \wedge \#_R(w') < \#_R(w))$$

We split $\mathbf{Q}(w)$ into $\mathbf{Q}^1(w)$ and $\mathbf{Q}^2(w)$ according to the two disjuncts: $\mathbf{Q}^1(w) = \{w' \mid (6) \wedge (7) \wedge \#_L(w') < \#_L(w)\}$. From (7), we learn that $\#_R(w')$ has to be equal to $\#_R(w)$. (And since w exists, this proves that $\#_R(w') < q$ holds.) Moreover, $\#_L(w')$ has to be equal to $\#_L(w) - 1$, implying $\#_L(w) \geq 1$. (Otherwise, we get a *hard* bottom since this property does not depend on the execution order.) Thus, the first intermediate result is:

$$K^1(w) = \textbf{if}\, w = L^{i\geq 1}.R^{j\geq 0}\, \textbf{then}\, L^{i-1}.R^j\, \textbf{else}\, \underline{\perp} \quad (9)$$

In a second step, $\mathbf{Q}^2(w) = \{w' \mid (6) \wedge (7) \wedge \#_L(w') = \#_L(w) \wedge \#_R(w') < \#_R(w)\}$. Equations (7) and $\#_L(w') = \#_L(w)$ cannot simultaneously be satisfied, so $K^2(w) = \perp$. Merging it with (9) yields the final result:

$$max(K^1(w), K^2(w)) = \left|\begin{array}{l}\textbf{if}\, w = L^{i\geq 1}.R^{j\geq 0}\\ \textbf{then}\, max\left(L^{i-1}.R^j, \perp\right)\\ \textbf{else}\, max\,(\underline{\perp}, \perp)\end{array}\right. = \left|\begin{array}{l}\textbf{if}\, w = L^{i\geq 1}.R^{j\geq 0}\\ \textbf{then}\, L^{i-1}.R^j\\ \textbf{else}\, \underline{\perp}\end{array}\right.$$

4.2 Second Example

We now study the program in Figure 2. Its simulated execution is given in Figure 4.2 so as to give the reader an intuitive feeling of how the flow of data behaves. Let $P(x) = \#_f(x) + \#_b(x)$.

Set of Possible Sources The set $\mathbf{Q}(w)$ of all previous instances w' of S that wrote in the memory cell read by w is:

$$\begin{aligned}\mathbf{Q}(w) = \{w' \mid \#_f(w') + \#_b(w') < N, & \quad (10)\\ \#_f(w') - \#_b(w') = \#_f(w) - \#_b(w) - 1, & \quad (11)\\ w' \ll w\} & \quad (12)\end{aligned}$$

Thanks to (1), (12) is equivalent to:

$$(w' = x.b.y'.S, w = x.S) \ \vee \ (w' = x.b.y'.S, w = x.f.y.S) \ \vee \ (w' = x.S, w = x.f.y.S) \quad (13)$$

where x, y, y' are in LABELS*. (10) is the existence predicate, (11) is the conflict predicate, i.e. the read element is the same as the one written into. We split $\mathbf{Q}(w)$ according to the disjuncts in (13).

Instance	write	read	source (absolute addr)	source (relative address)
$bbbS$	u^{-3}	u^{-4}	$\perp$	$\perp$
bbS	u^{-2}	u^{-3}	$bbbS$	$w.b.S$
$bbfS$	u^{-1}	u^{-2}	bbS	$w.f^{-1}.S$
bS	u^{-1}	u^{-2}	bbS	$w.b.S$
$bfbS$	u^{-1}	u^{-2}	bbS	$w.b^{-1}.f^{-1}.b.S$
bfS	ϵ	u^{-1}	$bfbS$	$w.b.S$
$bffS$	u	ϵ	bfS	$w.f^{-1}.S$
S	ϵ	u^{-1}	$bfbS$	$w.bfb.S$
$fbbS$	u^{-1}	u^{-2}	bbS	$w.b^{-2}.f^{-1}.b^2.S$
fbS	ϵ	u^{-1}	$fbbS$	$w.b.S$
$fbfS$	u	ϵ	fbS	$w.f^{-1}.S$
fS	u	ϵ	fbS	$w.b.S$
$ffbS$	u	ϵ	fbS	$w.b^{-1}.f^{-1}.b.S$
ffS	u^2	u	$ffbS$	$w.b.S$
$fffS$	u^3	u^2	ffS	$w.f^{-1}.S$

Fig. 3. Simulated execution for $N = 4$.

First Disjunct in (13)

$$\mathbf{Q}^1(w) = \{w' \mid (10), (11), w' = w.S^{-1}.b.y'.S\} \tag{14}$$

This is a simple NOL. (10) and (14) imply $P(w) < N - 1$, otherwise, $w' = \perp$. From (11) and (14), we have $\#_f(y') - 1 - \#_b(y') = -1 \;\Leftrightarrow\; \#_f(y') = \#_b(y')$. Let a be the number of f's and b's in y' ($a = \#_f(y') = \#_b(y')$). From (10) and (14): $\#_f(w) + 0 + a + \#_b(w) + 1 + a < N \;\Leftrightarrow\; P(w) + 1 + 2a < N$. Thus:

$$\mathbf{Q}^1(w) = \left| \begin{array}{l} \textbf{if } P(w) < N - 1 \\ \textbf{then } w.S^{-1}.b.\frac{[f^a, b^a]}{P(w)+1+2a<N}.S \\ \textbf{else } \emptyset \end{array} \right. \tag{15}$$

Let $a = max\left\{x \,\middle|\, x \in Z \;\wedge\; x < \frac{N-P(w)-1}{2}\right\}$. Then:

$$K^1(w) = \textbf{ if } P(w) < N - 1 \textbf{ then } w.S^{-1}.b.f^a.b^a.S \textbf{ else } \perp \tag{16}$$

Second Disjunct in (13) $\mathbf{Q}^2(w) = \{w' \mid (10), (11), w' = x.b.y'.S, w = x.f.y.S\}$. $w = x.f.y.S$ implies that w must contain at least one f. Moreover, $w' = w.S^{-1}.y^{-1}.f^{-1}.b.y'.S$. Thus, we have to consider the set of all $w.S^{-1}.\textsc{up}^*.f^{-1}$:

$$\mathbf{Q}^2(w) = \left| \begin{array}{l} \textbf{if } w = b^*.S \\ \textbf{then } \emptyset \\ \textbf{else } \left\{ w' \left| \begin{array}{l} \#_f(w') + \#_b(w') < N, \\ \#_f(w') - \#_b(w') = \#_f(w) - \#_b(w) - 1, \\ w' = w.S^{-1}.\textsc{up}^*.f^{-1}.b.\textsc{labels}^*.S \end{array} \right. \right\} \end{array} \right.$$

Lemma 2 implies that the last executed element in this set is the one with rightmost application of f^{-1}. I.e., the last one has the following shape: $w' = w.S^{-1}.b^{-k}.f^{-1}.b.y'.S$, where $k \geq 0$ is the greater value such that $w = \text{LABELS}^*.f.\ b^k.S$, and some word y'. (To see this, notice that Lemma 2 implies that $w.S^{-1}.f^{-1}.\ b^{-k}.f^{-1}.b.\text{LABELS}^*.S \ll w.S^{-1}.f^{-1}.b.\text{LABELS}^*.S$.) Let F, B be the number of f, b in y'. From (c): $B-F = k-1$.

$$K^2(w) = \left| \begin{array}{l} \textbf{if } w = b^*.S \\ \textbf{then } \perp \\ \textbf{else} \left| \begin{array}{l} \textbf{if } w = \text{LABELS}^*.f.b^k.S,\ \ k \geq 0 \\ \textbf{then } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \\ \textbf{else } \perp \end{array} \right. \end{array} \right.$$

Actually, we can't have $w \neq b^*.S \wedge w \neq \text{LABELS}^*.f.b^k.S$, so the last leaf is impossible:

$$K^2(w) = \left| \begin{array}{l} \textbf{if } w = b^*.S \\ \textbf{then } \perp \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right. \tag{17}$$

Note that, in the second leaf, w has to have shape $w = \text{LABELS}^*.f.b^k.S,\ \ k \geq 0$. Moreover, this leaf actually is a set. Even though this may mean some lack of precision in the analyis, this set is guaranteed to include *the* maximum element of $\mathbf{Q}^2(w)$ according to the lexicographical order.

Third Disjunct in (13) $\mathbf{Q}^3(w) : \{w' \mid (10), (11), w = w'.S^{-1}.f.y.S\}$. Let F, B be the number of f, b in y, respectively. From (11), we know that w cannot be equal to $b^*.S$. (However, since this is a property local to the current disjunct of the execution order, $w = b^*.S$ yields a *soft* bottom, not a hard one.) From $w = w'.S^{-1}.f.y.S$ and (11), $\#_f(w') - \#_b(w') = \#_f(w') + 1 + F - \#_b(w') - B \Leftrightarrow B - F = 0$. Thus w has to be of the form: $\text{LABELS}^*.f.[f^i, b^i].S$. Hence:

$$K^3(w) = \left| \begin{array}{l} \textbf{if } w = b^*.S \\ \textbf{then } \perp \\ \textbf{else} \left| \begin{array}{l} \textbf{if } w = \text{LABELS}^*.f.[f^i, b^i].S \\ \textbf{then } w.S^{-1}.[f^i, b^i]^{-1}.f^{-1}.S \\ \textbf{else } \perp \end{array} \right. \end{array} \right. \tag{18}$$

where i is a non-negative integer (= B = F). Lemma 2 implies that $i > j \Rightarrow w.S^{-1}.[f^{-i}, b^{-i}].f^{-1}.S \prec w.S^{-1}.[f^{-j}, b^{-j}].f^{-1}.S$, which in turns implies that the maximum of $w.S^{-1}.[f^{-i}, b^{-i}].f^{-1}.S$ is given by the smallest possible i, depending on the actual value of w.

Combining Intermediate Results Combining $K^3(w)$ and $K^2(w)$ ($K^3(w)$ is plugged in at $K^2(w)$'s leaves):

$$S_1(w) = \left| \begin{array}{l} \textbf{if } w = b^*.S \\ \textbf{then } \perp \\ \textbf{else} \left| \begin{array}{l} \textbf{if } w = \text{LABELS}^*.f.[f^i, b^i].S \\ \textbf{then } (a) \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right. \end{array} \right.$$

where:

$$(a) = max \begin{pmatrix} w.S^{-1}.[f^i, b^i]^{-1}.f^{-1}.S, \\ w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{pmatrix}$$

$$= \left| \begin{array}{l} \textbf{if } i = k = 0 \\ \textbf{then } w.S^{-1}.[f^i, b^i]^{-1}.f^{-1}.S \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right.$$

$$S_1(w) = \left| \begin{array}{ll} \multicolumn{2}{l}{\textbf{if } w = b^*.S} \\ \multicolumn{2}{l}{\textbf{then } \perp} \\ \textbf{else} & \left| \begin{array}{l} \textbf{if } w = \text{LABELS}^*.f.S \\ \textbf{then } w.S^{-1}.f^{-1}.S \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right. \end{array} \right.$$

Plugging $K^3(w)$ at S_1's leaves:

$$S_2(w) = \left| \begin{array}{ll} \multicolumn{2}{l}{\textbf{if } w = b^*.S} \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } P(w) < N-1 \\ \textbf{then } w.S^{-1}.b.f^a.b^a.S \\ \textbf{else } \perp \end{array} \right. \\ \textbf{else} & \left| \begin{array}{ll} \multicolumn{2}{l}{\textbf{if } w = \text{LABELS}^*.f.S} \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } P(w) < N-1 \\ \textbf{then } max(w.S^{-1}.f^{-1}.S, w.S^{-1}.b.f^a.b^a.S) \\ \textbf{else } w.S^{-1}.f^{-1}.S \end{array} \right. \\ \textbf{else} & \left| \begin{array}{l} \textbf{if } P(w) < N-1 \\ \textbf{then } max \begin{pmatrix} w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S, \\ w.S^{-1}.b.f^a.b^a.S \end{pmatrix} \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right. \end{array} \right. \end{array} \right.$$

Moreover, $w = y.f.s \;\Rightarrow\; w' = w.S^{-1}.f^{-1}.S \ll w' = w.S^{-1}.b.\text{LABELS}^*.S$, and $w = y.f.b^{k \geq 1}.S \;\Rightarrow\; w.S^{-1}.b^{-k}.f^{-1}.b.\text{LABELS}^*.S \ll w.S^{-1}.b.\text{LABELS}^*.S$, thus the two maxima should be replaced by their second arguments. Since we can then notice that all the leaves are equal when $P(w) < N-1$, we may factor the latter conditional out (just to make the result more readable to a human eye).

$$S_2(w) = \left| \begin{array}{ll} \multicolumn{2}{l}{\textbf{if } P(w) < N-1} \\ \multicolumn{2}{l}{\textbf{then } w.S^{-1}.b.f^a.b^a.S} \\ \textbf{else} & \left| \begin{array}{ll} \multicolumn{2}{l}{\textbf{if } w = b^*.S} \\ \multicolumn{2}{l}{\textbf{then } \underline{\perp}} \\ \textbf{else} & \left| \begin{array}{l} \textbf{if } w = \text{LABELS}^*.f.S \\ \textbf{then } w.S^{-1}.f^{-1}.S \\ \textbf{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^F, b^B]}{B-F=k-1, B+F<N+k-P(w)}.S \end{array} \right. \end{array} \right. \end{array} \right. \tag{19}$$

Notice also that, since all the intermediate results have been combined, remaining soft bottoms became hard ones, meaning that the analysis is now able to guarantee that there is no source under the corresponding conditions.

4.3 Comments on the Examples. Applications

The reader may check that the ***closed form*** (19) expresses the flows of data tabulated in Fig 4.2. For instance $w = b.b.b.S$ corresponds to the second leaf ($\underline{\perp}$). $w = f.b.b.S \Rightarrow k = 2$ in the last leaf, so $w' = max(b.\frac{[f^F, b^B]}{B-F=1, B+F<3}.S) = b.b.S$. All "inner" nodes correspond to the first leaf: for instance, $w = S \Rightarrow a = 1 \Rightarrow w' = b.f.b.S$.

Application of our data flow analysis to program checking is clear: in the first example, undefined values are read during the first instance of the outer `explore` construct.

Applications to compiling and automatic parallelization are best illustrated by the first example. Due to the regularity of the data flow, we can apply the method in [5] to automatically find a scheduling function$\theta(i, j)$ such that $\theta(i, j) > \theta(i-1, j)$. A possible solution is $\theta(i, j) = i$, i.e., the program can be "wavefronted" along the right branches of the tree. More formally, all instances $\langle S, i, j \rangle$, for a given i and all j, can be executed simultaneously.

In turn, we can apply the methods in [6] to detect that only a *finite subset* of the data structure `tree` (declared to be infinite) has to be allocated at any time: exactly 2 branches of q elements of `tree` need to be allocated, and can be reused in turn. In a sense, we thus have new means to compute the extent of the data structure [7].

5 Conclusion

We presented a data flow analysis for a restricted class of recursive programs over recursive data structures. This restricted class include recursive exploration of commutative (e.g. arrays) and non commutative (e.g. trees) recursive data structures. These restrictions are in part due to the intrinsic difficulty of the subject, and in part to the way our framework is restricted to data flow problems that can be expressed as integer linear programming ones, enforcing for instance that recursion predicates should be conjunctions of affine (in)equalities.

Rinard and Diniz [8] proposed an analysis that detects commuting (so, parallel) computations in C++ programs that manipulate recursive data structures. Even though they handle more general structures than we do, their analysis is local (i.e., does not take global properties of the structure into account) and is based on semantical properties of a limited number of operators.

There are obviously numerous deficiencies in our analysis: our inability to directly handle programs written in usual Pascal-like imperative languages (the main problems being the extraction of the existence predicate and the detection of induction variables that access data structures); the approximation due to NOLs in the case of non commutative structures; the lack of general algorithms to compute lexicographical maxima of NOLs with pumping inverses; and, of course, our need for more of practical experience. All these issues will be addressed in future work. In addition, we intend to study cases when the existence of operations cannot be predicted at compile-time, i.e., to extend our framework along the lines of [9].

Acknowledgments The first two authors are partly supported by the CNRS. The second, in addition, by a *Training and Mobility of Researchers* Grant from the European Union. The third author is partly supported by the DFG project RecuR. The last two authors are, in addition, supported by a German-French *Procope* exchange programme.

We would also like to thank Karl-Filip Faxén, Paul Feautrier, Chris Lengauer and Björn Lisper for helpful comments on this material.

References

1. B. Liskov et al. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–578, August 1977.
2. A. Cohen, J.-F. Collard, and M. Griebl. Data flow analysis of recursive structures. Technical report, Laboratory PRiSM, University of Versailles, France, September 1996.
3. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
4. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.
5. P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.
6. V. Lefebvre and P. Feautrier. Storage management in parallel programs. Technical Report 16, Laboratory PRiSM, University of Versailles, France, June 1996.
7. B. Lisper and J.-F. Collard. Extent analysis of data fields. In B. Le Charlier, editor, *Proc. International Symp. on Static Analysis (SAS'94)*, volume 864 of *LNCS*, pages 208–222, Namur, Belgium, September 1994. Springer-Verlag.
8. M. C. Rinard and P. C. Diniz. Semantic foundations of commutativity analysis. In L. Bougé et al., editor, *Euro-Par'96. Parallel Processing*, volume 1123 of *LNCS*, pages 414–423, Lyon, France, August 1996. Springer-Verlag.
9. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, CA, July 1995.