A Parallelization Framework for Recursive Tree Programs

Paul Feautrier *

Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

Abstract. The automatic parallelization of "regular" programs has encountered a fair amount of success due to the use of the polytope model. However, since most programs are not regular, or are regular only in parts, there is a need for a parallelization theory for other kinds of programs. This paper explore the suggestion that *some* "irregular" programs are in fact regular on other data and control structures. We adduce as an example the *recursive tree programs*, for which we build a parallelization model and a dependence test.

1 A Model for Recursive Tree Programs

The polytope model [Len93,Fea96] has been found a powerful tool for the parallelization of array programs. A brief description of its basic concepts is given below. Programs which conform to this model use only `DO` loops and arrays with affine subscripts. The relevant entities (iteration domain, data space, execution order, dependences) of such programs can be modeled as Z-polytopes, i.e. as sets of integral points belonging to bounded polyhedra. Finding parallelism in such programs depends on our ability to answer questions about the associated Z-polytopes, for which task we can use well known results from mathematics and operational research.

The aim of this paper is to investigate whether there exists other program models for which we can devise an automatic parallelization framework. The answer is yes, and we give as an example the *recursive tree programs* which are defined in sections 1.4 and 1.5. The relevant framework is presented in section 2. In the conclusion, we point to unsolved problems for the recursive tree model, and we suggest a search for other examples of parallelization frameworks.

1.1 An Assessment of the Polytope Model

In the classical literature, programs are seen as built from *elementary* statements by *control* statements. In denotational semantics, for instance, the meaning of an elementary statement is a function, and control statements are explained

* Paul.Feautrier@prism.uvsq.fr

as rules for combining these functions to get the meaning of more complicated constructs. For instance, the meaning of the sequence operator (usually written `;`) is function composition, and the meaning of a loop is a fixpoint operator.

The main lesson of the polytope model is that this point of view is not suitable for discussing parallelism, mainly because many questions which one has to answer have no meaning in it. For instance, it makes no sense to ask if a statement can be executed in parallel with itself, or in which order statements inside a loop body are executed. The suitable level of abstraction is the *operation*, i.e. one execution of a statement. It makes sense to ask whether two operations can be executed in parallel, and, in a sequential program, operations are totally ordered.

The problem with operations is that they are too numerous to be handled in extension. For instance, a program running for 1" on a 1 Mflops machine (no big deal at the present time), generate 10^6 operations. It follows that operation sets must be handled in intention, i.e. that we must design a finite symbolic representation for them.

In the case of `DO` loop programs, operations are created by loop iterations. Operations can be named by giving the values of the surrounding loop counters, arranged as an *iteration vector*. These values are integers. Moreover, they must be within the loop bounds. If these bounds are affine forms in the surrounding loop counters and constant *structure parameters*, then the iteration vector scans the integer points of a polytope, hence the name of the model. The sequential execution order in a perfect loop nest is lexicographic ordering of iteration vectors. This definition can easily be extended [Fea88] to cope with imperfect loop nests and whole programs.

To achieve parallelization, one has to find subsets of independent operations. Two operations are in dependence if they access the same memory cell, one at least of the two accesses being a write. To make use of this definition, we must be able to relate an operation and the accessed memory cells. If the data structures are arrays, this is possible if subscripts are affine functions of iteration vectors. The dependence condition then translates into a system of linear equations and inequalities, whose unknowns are the surrounding loop counters. There is a dependence if and only if this system has a solution in integers. There are well known *dependence tests*, ranging from exact algorithms, like Integer Linear Programming, to approximations like the Banerjee tests (see e.g. [ZC91]).

In the polytope model, one can pursue the parallelization process much farther, by the use of such techniques as array dataflow analysis, scheduling and placement (see [Len93,Fea96] and the references therein). In this paper, we will limit ourselves to the simpler problem of constructing a dependence test for another type of programs.

Let us try to summarize our observations as a set of requirements for a parallelization framework.

1. We must be able to describe, in finite terms, the set of operations of a program. This set will be called the *control domain* in what follows. The control domain must be ordered.

2. Similarly, we must be able to describe a data structure as a set of *locations*, and a function from the locations to values.
3. We must be able to associate sets of locations to operations through the use of *address functions*.

We can then set up a dependence problem as a set of conditions for the existence of a dependence. Solving the problem will be either exhibiting one dependence, or proving that none exists. This solution can be exact or pessimistic. In case of doubt, the dependence test must decide there *is* a dependence. Any other behaviour may lead to the construction of non deterministics parallel programs.

Our aim here is to apply these prescriptions to the design of a parallelization framework for recursive tree programs.

1.2 Related Work

We follow here the discussion in [HHN94]. The analysis of programs with dynamic data structures has been carried mainly in the context of pointer languages like C. The first step is the identification of the type of the data structures in the program, i.e. the classification of the pointer graph. The main types are trees (including lists), DAG and cyclic graphs. This can be done by static analysis at compile time [GH96], or by asking the programmer for the information. In this paper, we will use the second solution, and we will restrict ourselves to the case where the data structures are trees.

The next step is to collect information on the possible values of pointers. This is done statement-wise in the following sense: the set of possible pointer values is associated not to a runtime operation but to a so-called *program point* i.e. to a statement. These sets will be called *regions* here, by analogy to the array regions which were introduced by Triolet [TIF86] in the polytope model. Regions are usually represented as *path expressions*, which are regular expressions on the name of structure members [LH88]. For a more precise representation, in which Kleene stars can be replaced by named counters, see [Deu94].

Now, a necessary (but not sufficient) condition for two statements to be in dependence is that two of their respective regions intersect, one of these at least corresponding to a write. We will see later that this method incurs a loss of information which may forsake parallelization in important cases. One of the contributions of this paper is to improve the precision of the analysis for a restricted family of recursive tree programs.

Another work [Coh98] extend the concept of array dataflow analysis to recursive array programs. One of our long term objectives is to merge these two threads of research into one unified framework.

1.3 Basic Concepts and Notations

We recall here some basic facts of the elementary theory of finite state automata and rational transductions. The reader is referred to [Ber79] for a more detailed

treatment. This section can be skipped at first reading and used as a reference whenever a new concept or notation is encountered.

A finite set A (an alphabet) being given, A^* is the set of words on A, including the zero-length word, ϵ. In the following we will not distinguish between a letter (an element of A) and the corresponding word of length 1. The dot (.) denotes concatenation, whose unit element is ϵ.

We will use $\mathbb{N}$ (the set of nonnegative integers) as our basic alphabet. It will be understood that in any actual application, the alphabet is some finite subset of $\mathbb{N}$. This will dispense us with explicitly stating the alphabet before each example or theorem.

A *finite state automaton* (fsa) is a finite labeled graph. Its vertices are called *states* and its edges are called *transitions*. Edges are labeled with (one letter) words or ϵ. Some states are called initial and others (not necessarily disjoint) are terminal. The set of terminal states of automaton a is denoted $\mathrm{term}(a)$.

To each path from an initial to a terminal state, one associates the word obtained by concatenating the edge labels in the order of path traversal. The set of words obtained in this way from automaton a is the regular language generated (or recognized) by the automaton, $\mathcal{L}(a)$. A regular language can also be represented as a regular expression: an expression built from the letters and ϵ by the operations of concatenation (.), union (+) and Kleene star. There are well known algorithms for going from one representation to the other.

An automaton may have inaccessible states; these states can be removed without modifying the generated language. This process is called *trimming*. From one fsa, one may generates many others by changing the set of initial or terminal states, then trimming. For instance, $c(s;)$ is deduced from c by using s as the unique initial state. $c(;t)$ has t as its unique terminal state. In $c(s;t)$, both the initial and terminal states have been changed. An automaton is empty if, after trimming, it has no state left.

The family of regular languages is closed under many operation, including concatenation, Kleene star, union, intersection and complementation.

A rational transduction is a relation on $\mathbb{N}^* \times \mathbb{N}^*$ which is defined by a *generalized sequential automaton* (gsa): a fsa whose edges are labeled by digrams. A digram is a pair whose elements are either a letter or ϵ. Consider the regular language $\mathcal{L}(h)$ generated by h as a fsa. The first projection $\pi_1(w)$ of a word on digrams w is obtained by concatenating the first element of each digram in w. The second projection, π_2 is defined in a similar way. The relation defined by h is [Niv68]:

$$R = \{\langle \pi_1(w), \pi_2(w) \rangle \mid w \in \mathcal{L}(h)\}.$$

In the following, we will no longer distinguish between an fsa and the language it generates, or between a gsa and the relation it defines.

The family of rational transductions is closed by

- inversion (simply reverse the elements of each digram),
- concatenation (to build $f.g$, connect the terminal states of f to the initial states of g by edges bearing the null digram $\langle \epsilon, \epsilon \rangle$.
- composition [EM65].

Many subfamilies of fsa and gsa have been defined in the literature: deterministic fsa, length preserving gsa, rational functions, etc. We will have no use for such special cases in this work.

1.4 The Control Domain of Recursive Programs

Let us consider the following contrived example (the language uses a C-like syntax for better understanding):

```
int tree val;
void foo(address I)                    void bar(address J))
{int x;                                {
1 : if(...) bar(I.3);                  3 : foo(J);
2 : if(...) x = val[I.1];
}                                      }

                    void main(void)
                    {
                    4 : foo([]);
                    }
```

All statements have been numbered for easier reference. The only operative statement is 2; the function calls are control statements. The discussion will be clearer if we insure that labels are unique in the whole program, but this restriction is not mandatory. `trees` and `addresses` will be discussed later.

In this paper, we will not analyse conditionals. When we say, for instance, that operations u and v are in dependence, it will mean that, whenever the predicates in the tests are such that u and v are executed, then they are in dependence.

How do we identify a particular execution of statement 2? We observe that `foo` must be called first by `main`. There may then be an arbitrary number of calls of `bar` then `foo`. Then the test in statement 1 must fail, and 2 is executed. We may record the succession of function calls leading to an instance of 2 as a string, each "letter" being the name of a function. However, this representation is not precise enough, since a function may be called several time in the same body. We will use instead the labels of statements as letters of *call strings*. In the above case, all these strings can be represented in compact form by the regular expression $4.(1.3)^*.2$. This is not a piece of luck. In fact, to each recursive program we can associate a finite state automaton. The states are the functions and the basic statements of the program. The state associated to `main` is initial and the states associated to basic statements are terminal. There is a transition from state p to q if p is a function and q is a statement occurring in p body. The transition is labeled by the label of q in the body of p.

The result of this construction is the *control automaton* of the original program. The control automaton of the example above is given in Fig. 1. Let s be a terminal state of a control automaton. A string which is accepted by s gives a

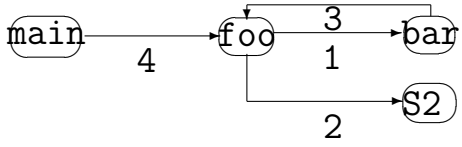

Fig. 1. A Control Automaton

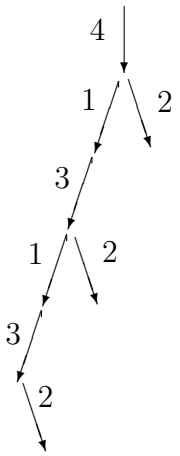

Fig. 2. A Control Tree

possible call sequence for the basic statement associated to s. The set of these strings (a regular language) is the control domain of the associated statement.

The reader is cautioned not to confuse a call sequence and execution history. Let us suppose for instance that the only information we have is that operations associated to the strings 4.1.3.1.3.2 and 4.2 have been executed. Both represent call sequences for statement 2. We can also say that 4.1.3.1.3.2 is executed before 4.2 because, in the body of the first call to `foo`, operation 1 and all the operations it initiates are executed before operation 2. However, we have no way of knowing whether operation 4.1.3.2, which would occur between the two distinguished operations, has been executed or not, since it is guarded by a test.

Notice also that the strings in the control domain can be arranged as a tree (see Fig. 2). This tree may be infinite, and is represented in finite terms by the associated control automaton. Each terminating execution corresponds to some finite subtree of the control tree.

As the example shows, if we take care of numbering statements in each function body by ascending numbers, the execution order is exactly lexicographic ordering on the call strings.

1.5 Addressing in Trees

Remark first that most tree algorithms in the literature are expressed recursively. Observe also that in the polytope model, the same mathematical object is used as the control domain and the set of locations of the data space. Hence, it seems quite natural to use trees as the preferred data structure for recursive programs.

In the case of ordered trees (trees in which the successor set of a given node is linearly ordered), there is a simple scheme for node naming. First, number all outgoing edges of a given node according to the order of the corresponding successors. The name of node n is then simply the string of edge numbers which are encountered on the unique path from the root of the tree to n. The name of the root of the tree is the zero-length string, ϵ. This scheme dates back at least to Dewey Decimal Notation as used by librarians the world over.

The set of locations of a tree structure is thus $\mathbb{N}^*$, and a tree object is a partial function from $\mathbb{N}^*$ to some set of values, as for instance the integers, the floating point numbers or the characters. More complicated tree types can be considered, but their study is left for future work. Furthermore, we will suppose that our data structures are trees of bounded fan-out. Hence, in fact, locations are strings on an initial segment of $\mathbb{N}$.

A tree in the above sense can be implemented as a set of C structures. Each structure corresponds to one tree cell. One member of the structure holds the value of the cell. Other members of the structure hold pointers to the successors of the cell.

Address functions map operations to locations, i.e. integer strings to integer strings. Furthermore, in a given program, the set of statements labels is finite. Hence, the functions we are interested in map words on some finite alphabet to words on another alphabet. Another point is that it is interesting to extend the concept of address functions to address *relations*. This allows us both to handle approximations — we do not know the exact address which is accessed by an operation, but we can exhibit a set to which that address belongs — and complex operations — e.g. operations which access all locations in a given subtree.

One family of relations which meets all these requirements is the set of *rational transductions* of [Ber79]. Consider again the above example. Notice the global declaration for the tree `val`. `address` is the type of integer strings. In line 2, such an address is used to access `val`. This address is built by postfixing the integer 1 to the value of the address variable `I`. This variable is initialized to ϵ at line 4 of `main`. If the call at line 1 of `foo` is executed, then a 2 is postfixed to `I`. `bar` does not change the value of its argument and use it directly to call `foo` again.

We may summarize this discussion by saying that at the entry to function `foo`, `I` comes either from line 3 or 4, which gives the following regular equation:

$$I = \langle 4, \epsilon \rangle + I.\langle 1, 2 \rangle.\langle 3, \epsilon \rangle,$$

whose solution is the regular expression:

$$I = \langle 4, \epsilon \rangle.(\langle 1, 2 \rangle.\langle 3, \epsilon \rangle)^*.$$

Lastly, the address which is used at line 2 is given by the following rational transduction : $\langle 4, \epsilon \rangle.(\langle 1, 2 \rangle.\langle 3, \epsilon \rangle)^*.\langle 2, 1 \rangle$. The reader may care to verify that, according to this formula, the address used by operation 4.1.3.2 is 2.1.

1.6 The $\mathcal{T}$ Language

We believe that the reasoning which has been used to find the above rational transduction can be automated, but the details have not been worked out yet. The technique seems to be very similar to those which are used for structure analysis, the difference being that one has to keep track of control information. For this "address analysis" to succeed we have to mimic the way a rational transducer works by appending one letter to the output each time an edge is traversed. This corresponds to the following class of address assignments:

```
<address variable> = <address_expression>
```

where an address expression is either an address constant (including `[]` which represents ϵ), an address variable or the result of postfixing an address variable by an integer constant.

The type of address variables is `address` in our language. An address constant is written as a list of integers separated by dots. If `t` is a tree and `a` is an address, then `t[a]` is the cell of `t` at address `a`. The zero-length address is written `[]`: see for instance statement 4 in the above example. The notation `t[[]]` (the root cell of `t`) may be abbreviated to `t[]`.

Besides these rules, our source language, $\mathcal{T}$ [1], will be like C, with the following restrictions:

- No pointers are allowed. In fact, addresses are to be used as a kind of disciplined pointers.
- The only data structures are scalars (integers, floats and so on) and trees thereof. Trees are always global variables. Addresses can only be used as local variables or functions parameters. No function may return an address. The semantics of parameter passing is the same as in C: copy in, no copy out.
- The only control structures are the conditional and the function call, possibly recursive. No loops or `goto` are allowed.

There remains the problem of tree construction. As a first approximation, we will suppose here that the first access to a tree cell creates it and all the not yet created cells from the new cell to the root. This is similar to the way array cells are created in Matlab. How to efficiently implement this proposal in a parallel context is left for future work.

Assessing the expressive power and ease of use of $\mathcal{T}$ is an open question. We have proved that one can simulate a Turing machine in $\mathcal{T}$, but the proof is omitted for lack of space. Some of the restrictions above can be removed by appropriate preprocessing. Loops, for instance, can be implemented as terminal recursions.

[1] $\mathcal{T}$ stands for "tree" and also for "toy".

2 Dependence Analysis of $\mathcal{T}$

2.1 Parallelization Model

When parallelizing static control programs, one has first to decide the shape of the parallel version. One can either construct processes, in which sequential constructions are executed in parallel, or fronts, which are parallel construction which are executed sequentially. In the later case, one usually distinguish between *control parallelism*, where the operations in a front are instances of different statements, and *data parallelism*, where a front is made of iterations of the same statement. It is common belief that there is more potential in data parallelism than in control parallelism, because there usually is much more data than statements in high performance programs.

In the case of recursive programs, it so happens that the distinction between control parallelism and data parallelism becomes moot. To see this, consider a piece of linear code:

```
{
S;
foo(x);
}
```

Suppose that we are able to decide that the operation associated to `S` and all operations generated by the call to `foo` are independent. We might rewrite the above sequence as:

```
{^
   S; foo(x);
^}
```

where we use the EARTH-C notation `{^ ... ^}` as the parallel counterpart of `{ ... }` [HTZ+97] (refer also to the Algol 68 `cobegin coend` construction). If `foo` is not recursive, then we have found a degree of parallelism of 2 (this can be improved by further analysis of `foo`, but it will stay bounded by the size of the program in any case). However, if `foo` is recursive and if in fact the above code is the body of `foo` itself, then the degree of parallelism is of the order of the number of recursive calls, which is data dependent. This is not surprising, since this example is a case of terminal recursion, which is known to be equivalent to a loop. In other situations (e.g. if a recursive function calls itself twice, and if the calls are independent), the amount of parallelism will be much larger (in this case, exponentially so).

To summarize our findings, we propose to parallelize recursive function bodies considered as linear sequences. A function is a candidate for parallelization if it is self-recursive or if it belongs to a recursion cycle. A possible formalization is the following.

Let us consider a function `foo` and the statements $\{S_1, \ldots, S_n\}$ of its body. The statements are numbered in textual order, and statement S_i is labelled i. For the purpose of this work, tests in a conditional statement are to be considered

as elementary statements, and must be numbered as they occur in the program text.

Let us construct a synthetic dependence graph (SDG) for `foo`. The vertices of the SDG are the statements of `foo`. There is a dependence edge from S_i to $S_j, i < j$ iff there exists three iteration words u, v, w such that:

- u is an iteration of `foo`.
- Both $u.i.v$ and $u.j.w$ are iterations of some terminal statements.
- Operations $u.i.v$ and $u.j.w$ are in dependence.

In the case where S_i and S_j are both elementary statements, the dependence may involve, not only tree cells, but also local variables and function parameters. Local variables are treated as scalars, hence the dependence calculation is trivial. Besides, one must not forget to add control dependences, from the test of each conditional to all statements in its branches. In the following, control dependences and dependences on local variables will be called classical dependences. Dependences on tree elements will be called tree dependences.

Once the SDG is computed, a parallel program can be constructed in several well known ways. The edges in the SDG can be interpreted as synchronization, with a `post` at the source of the dependence, and a `wait` at the sink. We will use here another possibility, which is to put the program in `fork ... join` form with the help of the topological sort algorithm. While this method may entail some loss of parallelism, the object programs look much better. As above, we will use the EARTH-C version of the `fork ... join` construct, `{^ ... ^}`. The run time exploitation of this kind of parallelism is a well known problem, see for instance [LGH97].

2.2 The Dependence Test

The first step of the analysis of the function `foo` is the construction of its classical dependence graph. This is a well known process, and we will not discuss it further here. The next step is, for each pair of statements $S_i, S_j, i < j$, to decide whether there exists a tree dependence. Notice that, at least in the present context, this is useless if S_i and S_j are already connected by a classical dependence.

Our main source of information is the control automaton, c. We have first to characterize, in the notations of section 1.3, the strings u which are iterations of `foo`. This is simply the language generated by $c(;\texttt{foo})$.

The string v above connect S_i to some terminal statement. Hence, it is generated by $c(S_i;)$. Notice that this automaton generate only ϵ when S_i is itself terminal. The terminal states of $c(S_i;)$ are candidates for the sources of tree dependences. Similarly, the terminal states of $c(S_j;)$ are candidates for the sinks of tree dependences. Given a pair of source and sink S_k and S_l, we can construct the associated automaton, $c(S_i; S_k)$ and $c(S_j; S_l)$. We may then construct the rational transduction which relates $x = u.i.v$ and $y = u.j.w$:

$$h = c(;\texttt{foo})^{=}.\langle i, j\rangle.c(S_i; S_k).c(S_j; S_l)^{-1}, \tag{1}$$

in which:

– if a is an automaton, then $a^{=}$ is the transduction obtained by setting each output word equal to the corresponding input word;

Refering back to the example in Sect. 1.4, the iteration words of function `foo` belong to the regular language $c(;\texttt{foo}) = 4.(1.3)^*$. One then has

$$c(;\texttt{foo})^{=} = \langle 4,4\rangle.(\langle 1,1\rangle.\langle 3,3\rangle)^*.$$

– an automaton can be used as a transduction whose output words have zero length. Similarly, the inverse of an automaton can be used as a transduction whose input words have zero length.

To each statement S_i are associated a list of read accesses, R_i and a list of write accesses, W_i. An access is composed of the name of a tree, t and of a rational transduction, f and is written $t[f]$. Given two accesses, one of them at least being a write, we have first to test if the accessed trees are the same, and then search for solutions of a problem in rational transductions. Before explaining how to solve this problem, let us summarize the preceding discussion by the following algorithm, which is the combinatorial part of the construction of the SDG:

Algorithm C.

Construct $c(;foo)$.
Forall $S_i, i = 1, n$
  Construct $c(S_i;)$
  Forall $S_j, j = i+1, n$
    Construct $c(S_j;)$
    Forall S_k among the terminal states of $C(S_i;)$
      Forall S_l among the terminal states of $C(S_j;)$
        Construct h according to (1).
        Forall $t[f] \in W_k$
          Forall $s[g] \in R_l$
            If $t = s$ then Dependence(f, g, h)
        Forall $t[f] \in W_k$
          Forall $s[g] \in W_l$
            If $t = s$ then Dependence(f, g, h)
        Forall $t[f] \in R_k$
          Forall $s[g] \in W_l$
            If $t = s$ then Dependence(f, g, h)

The dependence test We have to decide whether there exists three strings x, y, w such that:

$$\langle x, y\rangle \in h, \tag{2}$$
$$\langle x, w\rangle \in f, \tag{3}$$
$$\langle y, w\rangle \in g. \tag{4}$$

where (2) expresses the fact that x and y are iterations of S_k and S_l which are generated by one and the same call to `foo`, (3) and (4) expressing the fact that both x and y access location w.

The first step is to eliminate w, giving $\langle x, y\rangle \in k = g^{-1} \circ f$. k is a rational transduction by Elgot and Mezei's theorem [EM65]. We thus see that the pair $\langle x, y\rangle$ belongs to the intersection of the two transductions h and k. Deciding whether the intersection of two transductions is empty is a well known undecidable problem [Ber79]: Post correspondence problem can be reduced to it. Nevertheless, it is possible to define a semi-algorithm for solving it. Let us first introduce $\ell = k^{-1} \circ h$. Deciding whether $h \cap k$ is empty is clearly equivalent to deciding whether $\ell \cap =$ is empty (where $=$ is the equality relation, which is clearly a rational transduction).

Our semi-algorithm is best presented as a (one person) game in which the goal is to build a word u such that $\langle u, u\rangle \in \ell$. A position in the game is a tuple $\langle u, v, p\rangle$ where u and v are words and p is a state of ℓ. The initial position is $\langle \epsilon, \epsilon, p_0\rangle$, where p_0 is the initial state of ℓ. A move in the game consists in selecting a transition from p to q in ℓ with label $\langle x, y\rangle$. The outcome is a new position $\langle u', v', q\rangle$ where u' and v' are obtained from $u.x$ and $v.y$ by deleting their common prefix. A position is a loss if u and v begin by a different letter: in such a case, no amount of postfixing can complete u and v to equal strings. This leaves us with positions in which either u or v or both are ϵ. A position is a win if $u = v = \epsilon$ and $p \in \text{term}(\ell)$. Suppose now that $v \neq \epsilon$. Then, for success, v must be the beginning of a string in the domain of ℓ when starting from p. This can be tested easily, and, if the check fails, then the position is a loss. The situation is symmetrical if $u \neq \epsilon$.

This game may have three outcomes: if a win can be reached, then by restoring the deleted common prefixes, one reconstruct a word u such that $\langle u, u\rangle \in \ell$, hence a solution to the dependence problem. If all possible moves have been explored without reaching a win, then the problem has no solution. Lastly, the game can continue for ever. One possibility is to put an upper bound to the number of moves. If this bound is reached, one decides that, in the absence of a proof to the contrary, a dependence exists (this is an example of pessimistic k-limiting).

The following algorithm explores the game tree in breadth-first fashion.

Algorithm D.

1. Set $D = \emptyset$ and $L = \{\langle \epsilon, \epsilon, p_0\rangle\}$ where p_0 is the initial node of ℓ.
2. If $L = \emptyset$, stop. There is no dependence.
3. Extract the leftmost element of L, $\langle u, v, p\rangle$.
4. If $\langle u, v, p\rangle \in D$, restart at step 2.
5. If $u = v = \epsilon$ and if $p \in \text{term}(\ell)$, stop. There is a dependence.
6. If both $u \neq \epsilon$ and $v \neq \epsilon$, the position is a loss. Restart at step 2.
7. If $u = \epsilon$ and if v is not accepted by ℓ when starting from p, restart at step 2.
8. If $v = \epsilon$ and if u is not accepted by ℓ when starting from p, restart at step 2.
9. Add $\langle u, v, p\rangle$ to D. Construct all the positions which can be reached in one move from $\langle u, v, p\rangle$ and add them on the right of L. Restart at step 2.

Since the exploration is breadth-first, it is easy to prove that if there is a dependence, then the algorithm will find it.

Algorithms C and D have been implemented as a stand alone program in Objective Caml. The user has to supply the results of the analysis of the input program, including the control automaton, the address relations, the list of statements with their accesses and the classical dependences. The program then executes algorithms C and D, the result being the SDG of the program. All examples in this paper have been processed by this pilot implementation. As far as our experience goes, the case where algorithm D does not terminate has never been encountered.

2.3 A Simple Example

Let us consider the following $\mathcal{T}$ program.

```
#define BOOLEAN char

BOOLEAN tree leaf;
int tree value;
void sum(address I)
{
1 : if(! leaf[I]) {
2 :     sum(I.1);
3 :     sum(I.2);
4 :     value[I] = value[I.1] + value[I.2]
        }
}

void main(void) {
5 : sum([]);
}
```

`value` is a tree whose nodes hold integers, and `leaf` is a Boolean tree. The set $\{I \mid \texttt{leaf}[I] = 1\}$ is supposed to define a frontier of `value`. The problem is to sum up all integers on the frontier, the final result being found at the root of the tree (`value[]`).

The control automaton of this program is given by the following regular expression:

$$c = 5.(2 + 3)^*.(1 + 4).$$

while an analysis along the lines of Sect. 1.5 shows that the rational transductions associated with expressions `I`, `I.1` and `I.2` in statement 4 are respectively:

$$\text{here} = \langle 5, \epsilon \rangle.(\langle 2, 1 \rangle + \langle 3, 2 \rangle)^*.\langle 4, \epsilon \rangle \tag{5}$$

$$\text{left} = \langle 5, \epsilon \rangle.(\langle 2, 1 \rangle + \langle 3, 2 \rangle)^*.\langle 4, 1 \rangle \tag{6}$$

$$\text{right} = \langle 5, \epsilon \rangle.(\langle 2, 1 \rangle + \langle 3, 2 \rangle)^*.\langle 4, 2 \rangle \tag{7}$$

Let us consider the problem of parallelizing the body of `sum`. There are already control dependences from statement 1 to 2, 3, and 4. The crucial point is to prove that there are no dependences from 2 to 3. According to algorithm C, we have to consider two iterations of 4 (since the other basic statement, 1, does not access `value`) one from an iteration of 2, and one from an iteration of 3, and check whether they can access the same cell of `value`. There are one (candidate) output dependence from `value[I]` to itself, two flow dependences from `value[I]` to `value[I.1]` and `value[I.2]`, and two symmetrical anti-dependences.

Let us consider for instance the problem of the flow dependence from `value[I]` to `value[I.1]`. The related transduction begins in the following way:

$$\ell = (\langle 2,2\rangle + \langle 3,3\rangle)^*.\langle 3,2\rangle \ldots .$$

Algorithm D finds readily that there is no way of crossing the $\langle 3,2\rangle$ edge without generating distinct strings. Hence, there is no dependence.

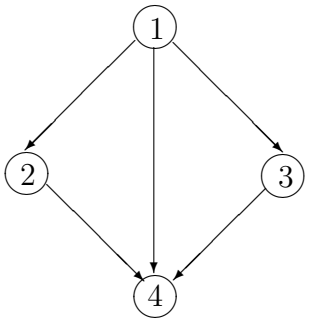

Fig. 3. The SDG of `sum`

Consider now the dependences from e.g. 2 to 4. Among the problems that have to be solved is the question of a flow dependence from `val[I]` to `val[I.1]`. Here we have:

$$\begin{aligned} f &= \langle 5,5\rangle.(\langle 2,1\rangle + \langle 3,2\rangle)^*.\langle 4,\epsilon\rangle,\\ g &= \langle 5,\epsilon\rangle.(\langle 2,1\rangle + \langle 3,2\rangle)^*.\langle 4,1\rangle,\\ h &= \langle 5,\epsilon\rangle.(\langle 2,2\rangle + \langle 3,3\rangle)^*.\langle 2,4\rangle.(\langle 2,\epsilon\rangle + \langle 3,\epsilon\rangle)^*.\langle 4,\epsilon\rangle \end{aligned}$$

Algorithm C finds the solution $x = 5.2.4$ from which follows $y = h(x) = 5.4$, then $w = f(x) = g(y) = 1$. All in all, the SDG of `sum` is given by Fig. 3, to which corresponds the parallel program:

```
void sum(address I)
{
```

```
1 : if(! leaf[I]) {
        {^
2 :       sum(I.1);
3 :       sum(I.2)
        ^};
4 :     value[I] = value[I.1] + value[I.2];
        }
}
```

a typical case of parallel divide-and-conquer.

3 Conclusion

3.1 Contributions

We have presented here a new framework in which to analyze recursive tree programs. The main differences between our method and the more usual pointer alias analysis are:

- Our data structures are restricted to trees, while in alias analysis, one has to determine the *shape* of the structures. This is a weakness of our approach.
- In our language, the operations on addresses are limited to postfixing, which, translated in the language of pointers, correspond to the more usual

  ```
  p = p -> member;
  ```

- Since our analysis takes into account all functions in a program, we do not have to use so-called handles [HTZ+97]. In fact, the only handle we use is the root of a tree.
- Our analysis is operation oriented, meaning that we associate set of addresses to operations, not to statements. This allows us to get more precise results when computing dependences.

In our formalism, we can reconstruct the usual alias analysis in the following way. Observe that, in the notations of Sect. 2.2, the iteration word x belongs to $\mathrm{Domain}(h)$, which is a regular language, hence w belongs to $f(\mathrm{Domain}(h))$. This is the *region* associated so S_k. Similarly, w is in the region $g(\mathrm{Range}(h))$. It follows that w belong to both regions, and hence to their intersection. The image of a regular language by a rational transduction is a regular language, and the intersection of two regular languages can be easily computed. When compared to our approach, the advantage of alias analysis is that the regions can be computed *a priori*, and that the emptiness or not of their intersection can be decided without reference to the control automaton.

It is easy to prove that when $f(\mathrm{Domain}(h)) \cap g(\mathrm{Range}(h)) = \emptyset$ then ℓ is empty. It is a simple matter to test whether this is the case. Our implementation reports the number of cases which can be decided by testing for the emptiness of ℓ, and the number of cases where Algorithm D has to be used.

In the case of `sum`, no dependence test can be solved by region intersection. This may seem counter-intuitive. The explanation is that any *specific* instance of the dependence problem *can* be solved by region intersection. For instance, one can prove in this way that the two topmost calls to `sum` are independent. We need the full force of Algorithm D to prove that calls to `sum` in *any* instance of `sum` itself are independent.

In a $\mathcal{T}$ implementation of the merge sort algorithm, there were 208 dependence tests. Of these, 24 were found to be actual dependences, 34 where solved by region intersection, and 150 required the use of algorithm D. While extrapolating from this example alone would be jumping at conclusions, it gives at least an indication of the relative power of the region intersection and of algorithm D. Incidentally, the SDG of merge sort was found to be of the same shape as the SDG of `sum`, thus leading to another example of a divide-and-conquer parallelization.

3.2 Further Problems

A first class of problems deals with the $\mathcal{T}$ language as it stands. One clearly need a sequential compiler, and a tool for the automatic construction of address relations. Some of the petty restrictions of Sect. 1.6 can probably be removed without endangering dependence analysis. For instance, having trees of structures or structure of trees poses no difficulty. Allowing trees and subtrees as arguments to functions would pose the usual aliasing problems. A more useful extension would be to allow trees of arrays, as found for instance in some versions of the adaptive multigrid method. Another set of questions is related to our parallelization model. Is it the best one? Can one build a theory of program transformations? What of memory expansion? Is there an equivalent of scheduling for $\mathcal{T}$ programs?

The second question has to do with the status of $\mathcal{T}$. Is it to be another programming language, or is it better used as an intermediate representation when parallezing pointer programs as in C or ML or Java? This would raise the question of translating C (or a subset of C) to $\mathcal{T}$, i.e. translating pointer operations to address operations. $\mathcal{T}$ is static with respect to the underlying set of locations. It is not possible, for instance, to insert a cell in a list, or to graft a subtree to a tree. How do we deal with this problem?

Thirdly, trees are only a subset of the data structures one encounter in practice. We envision two ways of dealing, e.g., with DAGs and cyclic graphs. One is to add new address operators to the language. For instance, adding a prefix operator:

$$\pi(a_1 \ldots . a_n) = (a_1 \ldots . a_{n-1})$$

allows one to handle doubly linked lists and trees with an upward pointer. The other possibility is to use other mathematical structures as a substrate. Finitely presented groups come immediately to mind, but there might be others.

References

[Ber79] Jean Berstel. *Transductions and Context-Free Languages.* Teubner, Stuttgart, 1979.

[Coh98] Albert Cohen. Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *TSI*, 1998. To appear.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI'94*. ACM Sigplan, 1994.

[EM65] C.C. Elgot and J.E. Mezei. On relations defined by generalized finite automata. *IBM J. of Research and Development*, pages 47–68, 1965.

[Fea88] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103. Springer-Verlag, 1996.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag or a cyclic graph? In *PoPL'96*, pages 1–15. ACM, 1996.

[HHN94] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general dependence test for dynamic, pointer based data structures. In *PLDI'94*, pages 218–229. ACM Sigplan, 1994.

[HTZ+97] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Shereen Ghobrial, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. *Int. J. of Parallel Programming*, 25(4):305–338, August 1997.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93, LNCS 715*, pages 398–416. Springer-Verlag, 1993.

[LGH97] Christian Lengauer, Sergei Gorlatch, and Christoph A. Herrmann. The static parallelization of loops and recursions. *J. Supercomputing*, 11(4):333–353, December 1997.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI'88*, pages 31–34. ACM Sigplan, 1988.

[Niv68] Maurice Nivat. Transductions des langages de Chomsky. *Ann. de l'Inst. Fourier*, 18:339–456, 1968.

[TIF86] Rémi Triolet, François Irigoin, and Paul Feautrier. Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Bernard Robinet and R. Wilhelm, editors, *ESOP 1986, LNCS 213*. Springer-Verlag, 1986.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM Press, 1991.