Resources Control Graphs

Jean-Yves Moyen

June 15, 2006

Resources Control Graph are a tool to perform some kind of abstract interpretation of programs in order to study their properties such as termination or complexity. The key idea is to work over the Control Flow Graph of a program and add some kind of dynamic information to it that can be seen as a way to approximate the different configurations reached during computation. Thus, any actual computation can be followed on the Resources Control Graph and studying properties of the Resources Control Graph gives information about the properties of the program itself.

Let us consider a program in some kind of imperative language that we won't need to describe in details here. In order to build its Control Flow Graph (CFG), we start by finding some control points in the program. Then, the CFG is a directed graph with one vertex per control point and one edge between a and b if there exists a computation of the program where the control flow goes between the control point corresponding to a to the control point corresponding to b. In most cases, that simply means that each control point is linked to the following one except for tests where control can go to either branch.

In order to illustrate this, let us consider as a toy example the following program, computing addition of two unary numbers and quite similar to what can be done with a counter machine:

$$\begin{array}{lcl} 0 & : & \texttt{if } x = 0 \texttt{ jmp } 4; \\ 1 & : & x--\ ;\ 2\ :\ y++; \\ 3 & : & \texttt{jmp } 0\ ;\ 4\ :\ \texttt{end}; \end{array}$$

Its CFG is displayed on Figure 1. Here, the control points correspond to the labels of the program. The useful property of CFG is that any execution of the program can be mapped onto a path of the CFG. The converse, however, is not true because paths of the CFG are somewhat non-deterministic in the sense that any branch of the test can be followed without taking into account the actual value of variables.

For example, the computation of $3 + 5$ corresponds to the path 012301230123`end`.

Any execution of the program can be seen as a sequence of states, each state containing both the current label and the current memory. A path in the CFG, on the other hand, only contain the sequences of vertexes, that is labels. Thus, the memory of the machine is completely forgotten in the CFG. Now, in order to study more precisely properties of programs, we want to add (part of) the forgotten information into the CFG, thus turning it into a *Resources Control Graph* (RCG).

So, instead of considering simply vertexes, we will now consider *configurations*. A configuration is a couple (v, x) where v is a vertex and x ranges over a (potentially infinite) set called the set of *valuations*. On the more general case, valuations represent exactly the current memory, but depending on the property under consideration, it is possible to restrict the set of valuations.

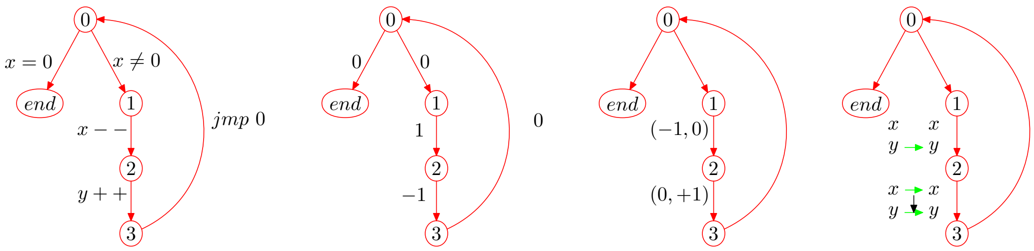

Figure 1: Control Flow Graph and Resources Control Graphes of the addition.

Now, the notion of path in the graph turns into a notion of *walks*. A walk is a sequence of configurations $(v_1, x_1), \dots, (v_n, x_n)$. The corresponding sequence of vertexes $v_1, \dots, v_n$ forms a path. Each new valuation x_{i+1} is computed from the preceding one and the edge followed by the walk.

So, to each edge of the RCG, that is to each instruction of the program, we need to define a function for computing new valuations when going through that edge. Here, again, depending on the approximation of memory chosen for the valuations, we may have different functions.

Let us illustrate this by considering only the total space usage of the program. Since our toy example works with unary numbers, the size needed to store a value n is n itself. So, at each state, the total space usage is $x + y$, the sum of the values of both registers. Since we chose this as an approximation of memory, valuations will be integers.

Now, we need to have functions associated to each edge that mimic, on the level of the approximation, the behaviour of the corresponding instruction. Both branches of the test, as well as the jump, keep the memory in the same state. Hence, the corresponding function is the identity. The $y++$ instruction increases the total space usage by one. Hence the corresponding function is $\lambda x.x + 1$. The $x--$ instruction decreases the total space usage by one *only if* $x \neq 0$ and causes an error otherwise. However, it is more convenient to consider that the decrease always occurs, that is associate to it the function $\lambda x.x - 1$, and carry the error on the level of the RCG.

Now, we can notice that all the functions associated to edges have the form $\lambda x.x + \alpha$ where α is some constant. Hence, it is more convenient to only represent them by the constant α. The corresponding RCG is thus a classical weighted graph as displayed on Figure 1.

Again, we have the same simulation property. That is, any execution of the program corresponds to a walk in the RCG. For example, the execution of $3 + 5$ corresponds to the walk (0, 8), (1, 8), (2, 7), (3, 8), (0, 8), ..., (0, 8), (`end`, 8).

But now, problems may occur because we've chosen to carry the errors of the subtraction on the RCG. Consider the following: $(0, 0), (1, 0), (2, -1)$. It is a walk that obviously cannot correspond to any execution because space usage cannot be negative. At this point, we would like the valuation to be restricted to only *positive* integers. However, this is not really satisfactory from a theoretical point of view since it is more convenient to have the set of valuations closed by the functions associated to each edge.

So, in order to circumvent this problem, we split the set of valuations and only consider some of them to be *admissible*. Similarly, a configuration will be admissible only if its valuation is admissible and a walk is admissible if all configurations in it are. Admissible valuations are those who really correspond to an approximation of the memory, not those added by the closure. Now, the previously mentioned walk is not admissible and can thus be banned.

The important point is that executions of the program are now mapped onto *admissible walks* only. So by only considering admissible walks, we do not miss anything. Of course, there is still the problem of detecting non-admissible walks.

Let us have a more closer look at the set of admissible walks in our case. Since the RCG does not contain any cycle of strictly positive weight (this is decidable in polynomial time via Bellman-Ford algorithm), any configuration (v, x) reached in a walk starting from (v_0, x_0) will have $x \leq x_0$. We say that the RCG is *resource aware*. Since, when mapping an execution of the program onto a walk of the RCG, the valuation corresponds to the total space usage, this means that when executing the program, one will never need more space than what is initially allocated. That is, the program is *Non-Size Increasing* [Hof99].

Of course, only considering the total space usage is a big approximation performed on memory. One may want to do something closer to the actual memory. Typically, one may want to consider the space usage of each variable independently. In this case, instead of a single integer, the valuations will range over vectors of integers with as many components as the number of variables, given a suitable enumeration of the variables. In our case, there are two variables and we will consider the vector $(|x|, |y|)$.

What will be the weight corresponding to the instructions in this case? Again, both branches of the test and the jump do not change the variables and hence have the identity as a weight. The $x--$ instruction decrease the size (value) of the first component without changing the size of the second component, hence its weight is $\lambda x.x + (-1, 0)$ (where vector-addition is performed component-wise). Similarly, the $y++$ instruction has weight $\lambda x.x + (0, +1)$. Again, since weights all have the form $\lambda x.x + \alpha$ for some constant α, we can identify them with α and the resulting RCG is a Vector Addition System with States (VASS) [Reu89]. The VASS for the addition is displayed on Figure 1.

Once more, every execution of the program can be mapped onto an admissible walk of the VASS. Here, the set of admissible valuations is exactly $\mathbb{N}^n$. For example, the execution of $3 + 5$ corresponds to the walk (0, (3, 5)), (1, (3, 5)), (2, (2, 5)), (3, (2, 6)), (0, (2, 6)), ..., (2, (0, 7)), (3, (0, 8)), (0, (0, 8)), (`end`, (0, 8)).

But now, if we look at the VASS, we can see that it admits no infinite admissible walk. The existence of an infinite admissible walk for a VASS is a decidable property. Since every execution is mapped onto an admissible walk, the absence of infinite admissible walk means that there are no infinite executions of the program, that is it is uniformly terminating.

However, using VASS as RCG to model programs is still not very convenient because one cannot model that way several useful instructions. Among other things, an usual copy instruction like $x := y$ cannot be represented that way. However, keeping the idea of having vectors representing the size of each variable as valuations, it then becomes quite easy to perform such copies by a matrix multiplication.

This is a rather appealing direction since matrices multiplications have already been used to study termination by Abel and Altenkirch [AA02] in their re-reading of the Size Change Principle [LJBA01]. And actually, it turns out that the SCP can indeed be expressed in terms of RCG where the weight are the size change graphs. The corresponding RCG for our addition program is depicted on Figure 1. The resulting termination criterion is equivalent to the so called "graph algorithm" of Lee, Jones and Ben Amram.

Using Abel and Altenkirch ideas, one can also express each size change graph (in the RCG) by a matrix. This leads to some kind of Matrices Multiplication Systems with States. But matrices multiplications has already been used by Niggl and Wunderlich [NW06] in order to detect polynomial time computations. Apparently, Niggl and Wunderlich criterion should also be expressable in terms of RCG.

So, the main interest of RCG is to be able to express within the same formalism several different existing program analysis. Moreover, some program transformation techniques such as the deforestation also seem to be easy to represent using Resource Control Graphs. This might lead to the building of some kind of global tool to perform programs analysis.

References

[AA02] A. Abel and T. Altenkirch. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[Hof99] M. Hofmann. Linear types and Non-Size Increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.

[LJBA01] C. S. Lee, N. D. Jones, and A. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL'01*, volume 28, pages 81–92. ACM press, January 2001.

[NW06] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*, 35(5):1122–1147, March 2006. published electronically.

[Reu89] C. Reutenauer. *Aspects mathématiques des réseaux de Petri.* Masson, 1989.