Control-Flow Refinment via Partial Evaluation

Jesús Doménech[1], Samir Genaim[2], and John P. Gallagher[3]

1 **Universidad Complutense de Madrid, Spain**
`jdomenec@ucm.es`
2 **Universidad Complutense de Madrid, Spain**
`sgenaim@ucm.es`
3 **Roskilde University, Denmark and IMDEA Software Institute, Spain**
`jpg@ruc.dk`

1 Introduction

Control-flow refinement is a technique used in program analysis to make the implicit control-flow of a given program explicit. Typically, this is done to increase the precision of the corresponding analysis. Consider for example the program on the left:

```
while ( x > 0 )
  if ( y < z ) y++; else x--;
```

```
while (x > 0 && y < z) y++;
while (x > 0 && y >= z) x--;
```

Its execution has two implicit phases: in the first one, variable y is incremented until it reaches the value of z, and in the second phase the value of x is decremented until it reaches the value 0. Control-flow refinement techniques can transform this program into the one on the right, in which the two phases are explicit.

In the context of termination analysis, such a transformation simplifies the termination proof; while the original program requires a lexicographic termination argument, the transformed one requires only linear ones. In addition, in the context of resource usage (cost) analysis, tools that are based on bounding loop iterations using linear ranking functions fail to infer the cost of the first program, while they succeed in inferring a linear upper-bound on the cost of the second one. In general, splitting the control-flow might also help in inferring more precise invariants, without the need for expensive disjunctive abstract domains, and thus improve any analysis that relies on such invariants (e.g. termination and cost analyses).

In the context of cost (and implicitly termination) analysis, control-flow refinement has been studied in [8] and [10]. The first [8] considers a general form of recurrence relations called cost equations, and the latter [10] considers structured imperative programs. Both handle programs with integer variables. These works demonstrated that control-flow refinement is crucial for handling programs that were considered challenging by the cost analysis community.

We started with the obvious observation that control-flow refinement is based on a special kind of partial evaluation tailored for a very particular analysis. We decided to explore what we would get, in terms of precision of cost and termination analysis, if instead, we use an existing off-the-shelf partial evaluation tool. This would allow integrating control-flow refinement into existing static analysis tools without any effort.

In this extended abstract, we describe preliminary experimental results obtained by using an off-the-shelf partial evaluation tool for control-flow refinement, and its impact on the precision of termination and cost analysis. Our experiments are done for Integer Transition Systems, which are graphs where edges are annotated with linear constraints describing transition relations between corresponding nodes.

2 Control-flow Refinement via Partial Evaluation for Cost and Termination Analyses

In this section we describe our experiments on the use of a partial evaluation tool for control-flow refinement, and its impact on the precision of termination and cost analysis.

Programs. Our programs are Integer Transition Systems. Briefly, such programs are described by control-flow graphs (`CFGs`) where edges are annotated with linear constraints over primed and unprimed integer variables. The unprimed variables represent the current state and the primed variables the next state; for example $\{x > 0, x' = x + 1\}$ describes the transition relation in which x must be positive to apply it, and the value of x increases by 1. We use this form since it is used as an intermediate representation in many termination and cost analyses.

Used Tools. For termination analysis we use `RankFinder` [2]. It is a termination analysis tool based on using linear and (several kinds of) lexicographic-linear ranking functions. It also infers supporting invariants using the abstract domain of polyhedra.

For cost analysis we use several tools: (1) `KoAT` [6], which is designed to analyze `CFGs` but does not use any control-flow refinement; (2) `PUBs` [4], which is the back-end of the `COSTA` [5] and `SACO` [3] systems. It accepts cost relations as input (a form of recurrence relation). We translate `CFGs` to this form when possible. (3) `CoFloCo`, which similarly to `PUBs` uses cost relations as input but it also applies control-flow refinement. We do not use or compare to the techniques of [10] since the corresponding tools are not available. For partial evaluation [11], we use a partial evaluation tool for constrained Horn clauses (CHCs) [12, 9] since `CFGs` can be seen as linear CHC programs. The partial evaluator yields a *polyvariant* specialisation, that it, it generates a finite number of versions of each predicate (where each predicate represents a program point), distinguished according to the constraints that hold upon reaching that point. Polyvariance is essential in order to achieve control-flow refinement.

In what follows, we first describe some selected interesting examples, and then describe and discuss our experiments on a larger set of examples.

2.1 Selected Examples

Iterative McCarthy 91

Our first example is the iterative McCarthy program depicted in Fig. 1. For an input $x > 100$ the loop body executes the *then* branch once and exits the loop; and for an input $x \leq 100$ the execution is done in two phases: in the first phase the *else* branch is executed repetitively until the value of x reaches the interval $x \in [101..111]$; and in the second phase the execution alternates between the *then* and *else* branches until c reaches 0. Note that it is guaranteed that c reaches 0 because executing the *then* and *else* branches consecutively increments x by 1 but does not change the value of c, and when reaching a state in which x is 111 the *then* branch is executed twice and thus the value of c is decreased by 1 (and x goes back to 91). The complexity of this program is $O(|x|)$; this is because for $x < 100$ the first phase executes at most $\frac{100-x}{11} + 1$ times, and the second phase executes at most $21(\frac{100-x}{10} + 1) + 1$ (because the value of c when entering the second phase is at most $\frac{100-x}{11} + 2$ and in the second phase it takes at most 21 iterations to decrease c by 1).

The `CFG` depicted in Fig. 1 (middle) corresponds to the iterative McCarthy program. It was obtained automatically using `llvm2KITTeL` [1]. Transition $\texttt{t1} = \{c > 0, c' = c, x' = x\}$ corresponds to entering the loop body, and transitions $\texttt{t3} = \{x > 100, x' = x - 10, c' = c - 1\}$

```
int McCarthy(int x){
    int c = 1;
    while (c > 0) {
        if (x > 100) {
            x -= 10;
            c--;
        } else {
            x += 11;
            c++;
        }
    }
    return x;
}
```

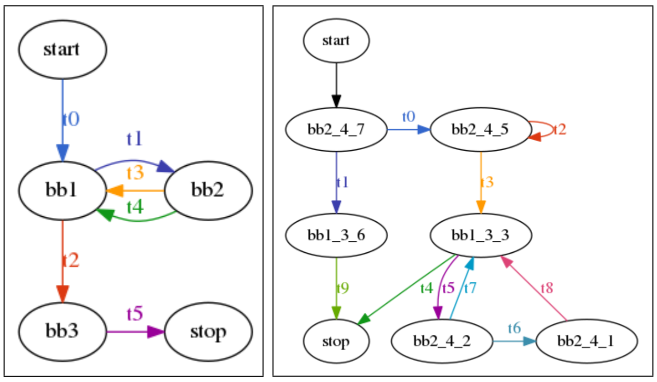

Figure 1 Iterative McCarthy 91

and `t4` $= \{x \leq 100, x' = x + 11, c' = c + 1\}$ to the *then* and *else* branches, respectively. Analyzing the termination behaviour of this `CFG` using `RankFinder` results in a termination proof with a lexicographical termination witness $\langle 10c - x + 90, x\rangle$ for the recursive SCC (that is defined by the nodes $\texttt{bb}_1$ and $\texttt{bb}_2$). Cost analysis using `KoAT` results in the bound $O(x^2)$, which is not precise enough. `CoFloCo` and `PUBs` were not able to infer any bound for this example. Recall that `CoFloCo` applies some control-flow refinement as well.

Applying `PE` on this `CFG` results in the `CFG` depicted Fig. 1 (right). Note that transition names in the two `CFGs` are not related. `PE` splits the control into two cases: one in which the input $x > 100$, which is represented by `t1` and `t9`, and one in which the input $x \leq 100$ which is represented by the rest of transitions. The subgraph that corresponds to the second case has 2 SCCs. The first one (node `bb2_4_5`) corresponds to the first phase in which the *else* branch is executed repetitively, and the second SCC (nodes `bb1_3_3`, `bb2_4_2` and `bb2_4_1`) corresponds to the second phase in which the *then* and *else* branches alternate.

For this `CFG`, `RankFinder` infers that first SCC terminates with a linear termination witness $100 - x$, and that the second terminates with a linear termination witness $335c - 6x$ as well (for each node there is a different free constant added to this function, but it is not important for our discussion). Having linear ranking functions makes cost analysis simpler as well, and indeed applying `KoAT` on this `CFG` we infer bound $O(x)$. Note that `PUBs` would infer a linear bound for this `CFG` as well since all SCCs are ranked by linear ranking functions.

Greatest Common Divisor

Consider the GCD program depicted in Fig. 2. It calculates the GCD of two positive integers using iterative subtracting. This program terminates and has a linear runtime complexity.

The `CFG` depicted in Fig. 2 (middle) is automatically obtained using `llvm2KITTeL`. The following are transitions that we will discuss later. `t3` $= \{x > 0, y > 0, x' = x, y' = y\}$, `t4` $= \{x > y, x' = x, y' = y\}$, `t4` $= \{x < y, x' = x, y' = y\}$, `t7` $= \{x < y, y' = y - x, x' = x\}$, and `t8` $= \{x \geq y, y' = y, x' = x - y\}$. Note that in `t8`, which corresponds to the *else* branch, we have the constraint $x \geq y$ while at runtime it will always be the case that $x > y$. This happens because `llvm2KITTeL` translates the if-statement independently from the context (the while-condition), and thus for the *else* branch it takes the negation of the if-condition.

`RankFinder` fails to prove termination of this `CFG`. The reason is that proving termination requires the invariant $\{x \geq 1, y \geq 1\}$ for node *bb*2 in order to rank transitions `t7` and `t8`.

```
int gcd(int x, int y){
 assert(x>0 && y>0);
 while(x != y){
   if(x<y)
     y = y-x;
   else
     x = x-y;
 }
 return x;
}
```

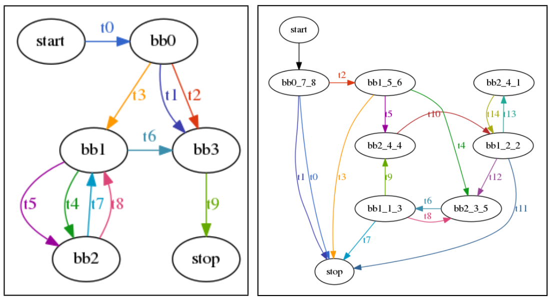

Figure 2 Greatest Common Divisor

RankFinder fails to infer this invariant mainly due to the constraint $x \geq y$, which at some point in the fixpoint computation introduces $x \geq 0$ at node $bb1$, and then the widening operation of PPL loses the lower bound of x (a more clever widening would have solved the problem). In order to solve this problem it is enough to unfold transitions in the recursive SCC, and thus collapse the two nodes into one (t_4 followed by t_7, and t_5 followed by t_8) which then will eliminate $x \geq y$. Note that unfolding is the basic operation in partial evaluation. KoAT and PUBs fail to infer a bound for this program, while CoFloCo infers the expected linear bound. Recall that CoFloCo applies control-flow refinment.

Applying PE to this CFG results in the one in Figure 2 on the right. We can see that PE actually did not collapse the node of the recursive SCC into one, which would have solved the problem. It actually split the two phases of the loop into separated ones and introduced transitions for moving from one to another – nodes bb1_2_2 and bb2_4_1 correspond to the *else* branch and nodes bb1_1_3 and bb2_3_5 correspond to the *then* branch. In addition, it has merged transitions so that the constraint $x \geq y$ disappeared.

Applying RankFinder of this CFG infers a linear ranking function for all components, in particular the nodes in the recursive SCC are annotated with the following function: bb1_1_3 : $\langle 4x + 2y - 1\rangle$, bb1_2_2 : $\langle 2x + 2y + 1\rangle$, bb2_3_5 : $\langle 2x + 2y\rangle$, bb2_4_4 : $\langle 4x - 2y\rangle$, bb2_4_1 : $\langle 2x + 2\rangle$. This is possible since now it infers the invariant $\{x \geq 1, y \geq 1\}$ where needed. Applying KoAT on this program we get the desired bound $O(|x|+|y|)$. PUBs would also get a linear bound for this CFG.

2.2 Preliminary Experimental Evaluation

We measured the precision of applying KoAT and RankFinder on two sets of examples, before and after partial evaluation. We preferred to use KoAT for cost analysis since it accepts CFGs as input without any need for reprocessing. We used two sets of integer transition systems: SET-A (416 benchmarks taken from the termination competition database) and SET-B (a set of 188 benchmarks used in [7] to evaluate the precision of CoFloCo).

The results of applying RankFinder on set SET-A (resp. SET-B) are: for 9 (resp. 7) CFGs using PE improves the result from *unknown* to *terminating*; for 26 (resp. 19) CFGs using PE improves the result from *lexicographic* to *linear* termination witness; for 7 (resp. 7) CFGs the analysis times out when analyzing the CFG generated by PE, since it was very large, while the analysis of the original CFG provides a proof of termination; for 3 (resp. 2) CFGs using PE results in a *lexicographic* termination witness while the original in a *linear* one; and for the rest of CFGs the results are equivalent.

The results of applying KoAT on set SET-A (resp. SET-B) are: for 72 (resp. 7) CFGs using PE improves from *unknown* to some upper-bound; for 0 (resp. 16) CFGs using PE improves the the complexity class of the upper-bound; for 10 (resp. 7) CFGs the analysis times out when analyzing the CFG generated by PE, since it was very large, while the analysis of the original CFG provides an upper-bound; for 19 (resp. 6) CFGs using PE results in an upper-bound of a worse complexity class; and for the the rest of CFGs the results are equivalent.

In summary, in many cases using PE improves the results of both termination and cost analyses. However, in some CFGs it generates large CFGs that the analysis is not able to handle, and in some other cases they lead to worse results. We still need to explore these (large) example in details in order to understand the reason.

3 Conclusions and Future Work

In this extended abstract we explored the use of PE as a control-flow refinement technique in the context for termination and cost analysis. Our preliminary experiments show that PE can improve both analyses. However, there are several cases where the results are worse. Currently we are investigating these examples to identify the reason for this imprecision.

Apart from this, for future work we plan to follow up these preliminary research results. In particular, we intend to pursue the following directions. (1) To measure the effect of applying PE on performance. In particular if we can use PE only for parts of the CFG (e.g., selected SCCs) for which the analyzer produces imprecise results; (2) to measure the effect of using PE on other termination and cost analysis tools (apart from KoAT and RankFinder); (3) to explore the use of PE for inferring lower-bounds; (4) to specialize the PE tool we use in order to take into account that is applied in the context of termination and cost analysis.

References

1 llvm2KITTeL. https://github.com/s-falke/llvm2kittel.

2 RankFinder. http://www.loopkiller.com:8081/ei/clients/web/.

3 E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.

4 E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

5 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. of FMCO'07*, volume 5382 of *LNCS*, pages 113–132. Springer, 2008.

6 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016.

7 A. Flores-Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Technische Universität, Darmstadt, August 2017.

8 A Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proc. of APLAS'14*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.

9 J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of the ACM SIGPLAN Symposium on PEPM'93*, pages 88–98, 1993.

10 S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM Sigplan Notices*, volume 44, pages 375–385. ACM, 2009.

11 N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

12 B. Kafle, J. P. Gallagher, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey. An iterative approach to precondition inference using constrained Horn clauses. *ICLP'18*, 2018.