The FuncTion Static Analyzer

Caterina Urban

Abstract FuncTion is a research prototype static analyzer designed for proving conditional termination and other CTL properties of C-like programs. The tool automatically infers sufficient preconditions for these properties by means of *abstract interpretation*. The underlying tunable abstract domain, based on piecewise-defined functions, offers a flexible balance between the precision and the cost of the analysis.

Key words: Static Analysis, Abstract Interpretation, Termination, Ranking Functions

1 Introduction

Software defects plague us in our daily life[1] and can have catastrophic consequences in safety-critical applications such as space[2], avionic or automotive[3] transportation and medical monitoring systems[4]. For this reason, increasing resources are devoted to specifying and verifying the behavior of computer programs.

For program specification purposes, Clarke and Emerson introduced *computation tree logic* (CTL) [7], which nowadays is one of the most well-studied and used temporal logics (together with LTL [30]). FuncTion is an open-source research prototype static analyzer that automatically infers sufficient preconditions for CTL properties of C-like programs. In particular, it is specialized in proving liveness properties — notably, conditional termination — for which it also infers valid ranking functions [33].

Caterina Urban
Inria & École Normale Supérieure | PSL University, Paris, France, e-mail: caterina.urban@inria.fr

[1] http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/
http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption/
http://heartbleed.com

[2] http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[3] http://www.automotive-spin.it/uploads/12/12W_Bagnara.pdf

[4] http://sunnyday.mit.edu/papers/therac.pdf

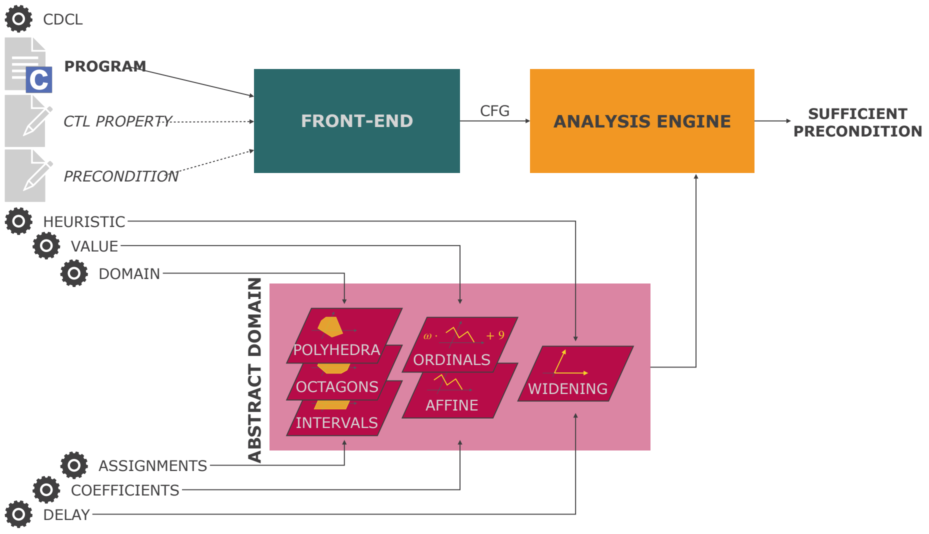


Fig. 1: Overview of FuncTion's architecture.

FuncTion is based on *abstract interpretation* [11], a general theory of semantic approximation at the basis of many successful industrial-scale tools (e.g., Astrée [4]). The underlying abstract domain is based on piecewise-defined functions of the program variables. Their domain of definition yields sufficient preconditions for the considered CTL properties, which thus gives more information that simply determining whether these properties are satisfied or not by the program. For liveness properties, the value of the piecewise-defined functions also yields an upper bound on the number of execution steps needed to reach the desired program state. Various features of the abstract domain are tunable to balance precision and cost of the analysis.

Outline The chapter is organized as follows. Section 2 gives an overview of FuncTion's architecture. Section 3 gives a glimpse of the theory behind the tool. Each component of FuncTion, from input to output, is presented in Sections 4 to 8. An example of conditional termination analysis performed by FuncTion is detailed in Section 9. Section 10 discusses a couple of dead ends encountered during the development of the tool. Finally, Section 11 concludes by discussing future work.

2 Tool Architecture

FuncTion consists of around 9K lines of OCaml code available on GitHub[5].

Figure 1 shows an overview of FuncTion's architecture. The tool takes as input a C-like program and, optionally, a CTL property of interest and a program precondition

[5] https://github.com/caterinaurban/function

(cf. Section 4). If no CTL property is provided, the tool proves conditional termination for the program. Without a precondition, the program is analyzed for all possible inputs. The front-end (cf. Section 5) takes care of parsing the program, building its CFG, and passing it to the right analysis engine (cf. Section 6), depending on the property of interest. The abstract domain (cf. Section 7) used by the analysis can be configured to use more or less precise domain and value representations (cf. Section 7.1 and 7.2) as well as more or less sophisticated widening heuristics to approximate loops (cf. Section 7.3). The tool outputs a sufficient precondition for the desired property which, for liveness properties, is also a valid ranking function (cf. Section 8). More details on each tool component and configuration are given in the referenced sections.

3 Behind the Scenes

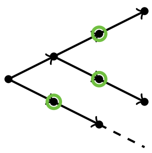
FuncTion is built upon the abstract interpretation framework for termination proposed by Cousot and Cousot [14] and later generalized to other liveness properties [38] and to all CTL properties [39].

The theory of abstract interpretation provides a uniform framework for systematically deriving a sound and complete proof method for a program property of interest [12]. Indeed, for reasoning about a particular property of a program, it is not necessary to consider all details of the program execution. On the contrary, reasoning is facilitated by abstracting away irrelevant matters, and modeling the behavior of the program with a semantics reduced to essentials to capture exactly and only the needed information. Therefore, there is no universal general-purpose program semantics but rather a wide variety of special-purpose semantics, each dedicated to a particular class of program properties. These can be uniformly defined as fixpoints of monotonic functions over ordered structured, and organized into a hierarchy of interrelated semantics specifying at various levels of detail the behavior of programs [9]. The correspondence between the semantics in the hierarchy is established by Galois connections formalizing the loss of information. Further sound abstractions suitable for static program analysis are then derived by fixpoint approximation [11].

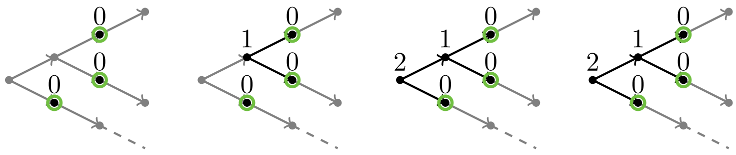
Following this framework, we systematically derived a sound and complete program semantics for any CTL property [39]. The semantics is a function defined over the program states that satisfy the given CTL formula. This choice yields a uniform treatment of CTL formulas independently of whether they express safety or liveness properties, or a combination of these [2]. For liveness properties, the semantics is their most precise *ranking function* [19, 33], mapping program states to elements of a well-ordered set whose value decreases during program execution. Specifically, our semantics maps program states to ordinals representing an upper bound on the number of program execution steps needed to reach a desired program state [14, 38].

We define the semantics inductively over the structure of a CTL formula, and we express it in a constructive fixpoint form starting from the semantics defined for its sub-formulas. Let us consider, for instance, the CTL property $\mathsf{AF}\varphi$, which states that in any program execution (A) eventually (F) a state satisfies φ. Intuitively, the program semantics for $\mathsf{AF}\varphi$ is defined starting from the program states that satisfy φ, where it has value zero, and retracting the program backwards while mapping each program state definitely leading to a state satisfying φ to an ordinal representing an upper bound on the number of program executions step needed until φ is satisfied. The domain of such a partial function w is the set of program states satisfying $\mathsf{AF}\varphi$: all program executions branching from a state $s \in \mathrm{dom}(w)$ lead to a state satisfying φ in at most $w(s)$ program execution steps, while at least one program execution branching from a state $s \notin \mathrm{dom}(w)$ never leads to a state satisfying φ.

Example 1 Example Let us consider the following depiction of program executions:



where arrows represent transitions between states (the dashed line represents an infinite execution) and highlighted states satisfy a formula φ. The program semantics for the CTL property $\mathsf{AF}\varphi$ is iteratively defined up to a fixpoint as follows:



where states are labeled with the corresponding value of the program semantics, and unlabeled greyed out states are outside the domain of the function.

This program semantics is sound and complete for proving any CTL property [39] but it is usually not computable. We thus compromise on completeness to derive a decidable sound approximation [34]. The abstract semantics over-approximates the value of the (concrete) program semantics and under-approximates its domain of definition. In this way, we infer sufficient preconditions for any CTL property: all program states in the domain of definition of the abstract semantics satisfy the CTL property, while other program states may or may not satisfy the CTL property.

More specifically, our abstract semantics is a *piecewise-defined function* mapping partitions of the program states to linear functions (or their lexicographic combinations [37]) of the program variables. The abstract semantics for any CTL formula builds upon the abstract semantics computed for its sub-formulas. It is automatically computed though a backward analysis of the program: its partitions are modified by program assignments and split by program path conditions, while (for liveness properties) its value is strictly increased by each program statement. To minimize the cost of the analysis and to enforce termination, a widening operator limits the number of maintained partitions. More details on the analysis and the abstraction are given in Section 6 and Section 7.

4 Tool Input

FuncTion expects as input a program written in a simple subset of C supporting local variables, function calls, and some jump statements.

More specifically, the syntax allows top level global variable declarations and function declarations. The only supported variable type is mathematical integers, deviating from the standard semantics of C. Variables are optionally initialized; without an explicit initializer they are implicitly initialized to 0. Functions take any number of integer arguments and return either an integer or no value.

Inside functions the allowed instructions are local variable declarations, assignments, function calls, test and loop control statements, jump statements, and blocks of instructions in curly brackets. Local variables are only visible inside the current block. Integer expressions on the right-hand side of variable assignments are built using arithmetic operators — addition, subtraction, multiplication, and division — on integers, and can contain function calls. Function calls can also appear as stand-alone instructions. The condition in a control statements must be a boolean expression, that is, either a comparison of integer expressions or a boolean combination (a conjunction or a disjunction) of expressions. The only allowed loop control statements are `while` and `for` loops. Finally, the supported jump statements are `break`, `return`, and `goto`. The return statement optionally takes an integer expression as parameter. The goto takes a label designating the destination of the jump. Both forward and backward jumps are allowed.

FuncTion, optionally, also takes as input a CTL formula specifying a property of interest and a program precondition. The syntax of CTL formulas is given by the following grammar:

$$\begin{aligned} \phi &::= a \mid \mathsf{NOT}\{\phi\} \mid \mathsf{AND}\{\phi\}\{\phi\} \mid \mathsf{OR}\{\phi\}\{\phi\} \mid \mathsf{A}\varphi \mid \mathsf{E}\varphi \\ \varphi &::= \mathsf{X}\{\phi\} \mid \mathsf{G}\{\phi\} \mid \mathsf{U}\{\phi\}\{\phi\} \end{aligned}$$

where a is an atomic boolean expression defined over the program variables (of the program state of interest). Instead, program preconditions are conjunctions of atomic boolean expressions. The CTL formula (or its absence) dictates the analysis run by the tool on the input program (cf. Section 6). A program precondition limits the analysis to only certain inputs.

5 Front-End

For parsing programs and CTL properties, FuncTion uses dedicated ad-hoc parsers generated using Menhir[6]. The program parser outputs its abstract syntax tree (AST) which is then transformed into a control flow graph (CFG).

The nodes in the CFG have a unique integer identifier and a reference to the corresponding position in the original source code. Unique entry and exit nodes are added for convenience. Nodes additionally maintain a list of incoming and outgoing edges.

[6] http://cristal.inria.fr/~fpottier/menhir/

The edges of the CFG are also uniquely identified; each connects two nodes in the CFG and is labeled by an instruction to be executed when going from the source to the destination node of the edge.

The syntax of these instructions is much simpler than the program abstract syntax. The supported instruction are skip statements, variable assignments, guards, and function calls. Skip statement do nothing and are used to model jumps or returns without a value. Integer expressions in variable assignments are simplified to not contain function calls; each call is replaced by the variable storing the return value of the function. Guards allow transitioning from the source to the destination node of an edge only if they evaluate to true. Finally, function calls merely indicate the name of the function being called; the AST to CFG transformation takes care of adding assignment storing actual arguments into formal arguments and storing the return value (if any) of the function.

Once the CFG is constructed, FuncTion's front-end runs a simple dominator-based loop-detection algorithm [1] to find loop heads. The performance of this algorithm is polynomial but proved sufficiently fast in our experience so far. Faster algorithm exist [26] should the need for them arise.

6 Analysis Engine

FuncTion's analysis engine walks over the given program CFG and computes an abstract information at each node, depending on the chosen analysis. All analyses are performed backwards starting from the exit node of the CFG using a standard worklist algorithm [29]. At each step, a CFG node is extracted from the worklist and its associated abstract value is updated. A new value for the node is computed by applying the instruction for each successor edge (that is, interpreting each instruction in the abstract domain, cf. Section 7) and joining the results. If the new value is different, then all predecessor nodes are put into the worklist. The analysis terminates once the worklist is empty. To enforce termination in the presence of loops, widening is used at loop heads. FuncTion can be configured to delay the widening for a number of iterations (parameter DELAY in Figure 1; by default the widening delay is set at two iterations).

Note that this simple iteration algorithm can often cause the tool to perform useless work (e.g., analyzing a path in the CFG with only a partial result that will later be overwritten), unless one is careful with the order and frequency in which nodes are inserted into the worklist. In our experience this had a negligible impact on performance. Should this become an issue, other iteration algorithms exist that do not suffer from it [5].

The initial abstract domain value and the interpretation of each instruction in the abstract domain is specific to the chosen analysis. For conditional termination (i.e., when no CTL formula is given in input to FuncTion, cf. Section 4), the analysis starts with an abstract domain value representing the constant function equal to zero; this denotes that the program has terminated. A guard partitions the domain of this function, strictly increases its value on the partitions that satisfy the guard and discards its value on the other partitions; the resulting abstract domain value denotes that the program will terminate for all variable values that satisfy the guard — the value increase ensures that the (ranking) function counts the program execution step. Instead, a variable assignment modifies the partitions and increases the value of the function on each of them (if any). We refer to [34] for more details.

When a CTL formula is given, the interpretation uses the result of the analysis performed recursively on each sub-formulas. For example, let us consider again the CTL property $\mathsf{AF}\varphi$. The analysis is first run for the sub-formula φ; this yields a map from each node in the CFG to an abstract value representing a partial function only defined for all variable values that satisfy φ. The analysis for $\mathsf{AF}\varphi$ then proceeds as for conditional termination except for two differences: (1) the initial abstract value represents the totally undefined function, and (2) at each node of the CFG, all partitions on which the abstract value for φ is defined replace those in the current abstract value; this resets the counter on the number of program execution steps needed until φ is satisfied. We refer to [39] for a formal definition of the analysis for each CTL formula.

Finally, if a program precondition is given, FuncTion runs a precursory forward reachability analysis. For this it, uses the same numerical domain chosen for the domain representation of the main abstract domain (cf. Section 7). The backward analysis then discards the value of the function associated with each node of the CFG for the unreachable partitions. This mechanism can be pushed even further by configuring FuncTion to recursively reanalyze the program on the partitions in which the analysis lost too much precision and could not prove the given property (parameter cdcl in Figure 1) [17].

7 Piecewise-Defined Functions Abstract Domain

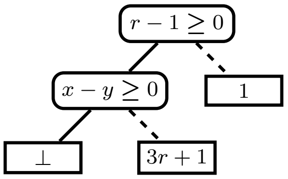
FuncTion's piecewise-defined functions abstract domain was first introduced in [34] and later extended in [37, 36, 8]. The abstract domain has also inspired recent work on the static analysis of software product lines [16].

Inspired by [4, 20, 24], the abstract domain elements are internally represented by *decision trees*, where the decision nodes are labeled by constraints over the program variables, and the leaf nodes are labeled by functions of the program variables. The decision nodes recursively partition the space of possible values of the program variables and, for liveness properties, the functions at the leaves provide the corresponding upper bounds on the number of program execution steps needed to reach the desired program state.

An example of decision tree is shown in Figure 2b. This is the decision tree produced by FuncTion for proving conditional termination of the loop in Figure 2a. At each loop iteration, the value of r is increased by the value of x and decreased by the value of y. Thus the loop terminates if and only if $x < y$. The linear constraints labeling the nodes of the tree are satisfied by their left subtree, while their right subtree satisfies their negation. The leaves of the tree represent partial functions the domain of which is determined by the constraints satisfied along the path to the leaf node. The leaf with

```
int f (int x, int y, int r) {
    while ( r > 0 ) {
        r = r + x;
        r = r − y;
    }
    return 0;
}
```

(a) Simple C function. It terminates if and only if $x < y$.



(b) Decision tree representation of the piecewise-defined partial function the domain of which represents a sufficient (and, in this case, also necessary) precondition for termination for the function f.

Fig. 2: Simple example of C function (a) and corresponding decision tree produced by FuncTion for proving its conditional termination (b).

value $\perp$ explicitly represents the undefined partition of the partial function represented by the decision tree. In this case, the decision tree indicates that, if $x < y$, the function f terminates in at most r loop iterations (i.e., in at most $3r + 1$ program steps).

7.1 Domain Representation

More technically, the decision nodes are labeled by *linear constraints* provided by the APRON numerical abstract domain library [25]. In order of expressiveness, FuncTion can be configured (parameter DOMAIN in Figure 1) to use interval constraints [10] (i.e., of the form $\pm x \leq c$), octagonal constraints [28] (i.e., of the form $\pm x_i \pm x_j \leq c$), or polyhedral constraints [15] (i.e., of the form $c_1 \cdot x_1 + \cdots + c_k \cdot x_k \leq c_{k+1}$).

These linear constraints are modified by the variable assignments encountered during the analysis. FuncTion supports two strategies for carrying out assignments (parameter ASSIGNMENT in Figure 1). The default strategy consist in traversing the decision tree and performing the assignment independently on each decision node [36]; this strategy is cheap but sometimes imprecise. A more precise but also costlier strategy consists in performing the assignment on each path of the decision tree (i.e., on each partition of the function represented by the tree) and then merging the resulting partitions [8]. This strategy requires traversing the initial decision tree to identify the partitions, building a decision tree for each resulting partition, and traversing these decision trees to merge them; note that building a decision tree requires sorting a number of constraints possibly higher than the height of the initial decision tree [35].

Interestingly, contrary to expectations, octagonal constraints are the costliest decision node labeling in practice. This is because both assignment strategies amplify known performance drawbacks for octagons [21]. Specifically, since octagonal constraint manipulation algorithms do not preserve their sparsity, performing a variable assignment on a single octagonal constraint often yields multiple linear constraints. This causes decision trees to grow considerably in size. The effect is particularly exacerbated by FuncTion's default assignment strategy.

7.2 Value Representation

The leaves of the decision trees are labeled by *affine functions* of the program variables (i.e., of the form $m_1 \cdot x_1 + \cdots + m_k \cdot x_k + q$), plus the special elements $\perp$ and $\top$ which explicitly represent undefined functions. The element $\top$ is only introduced by the widening operator to signal an unrecoverable loss of precision (cf. Section 7.3). More specifically, FuncTion supports *lexicographic* affine *functions* $(f_k, \ldots, f_1, f_0)$ in the isomorphic form of ordinals $\omega^k \cdot f_k + \cdots + \omega \cdot f_1 + f_0$ [27, 36]. The maximum degree k of the polynomial is a parameter of the tool (VALUE in Figure 1); a degree of 0 is equivalent to a single affine function.

Internally, the affine functions are actually polyhedral constraints over the program variables plus a special variable ν representing the value of the affine function, that is, an affine function $m_1 \cdot x_1 + \cdots + m_k \cdot x_k + q$ is actually a polyhedral constraint $\nu \leq m_1 \cdot x_1 + \cdots + m_k \cdot x_k + q$. This allows reusing the APRON polyhedra domain implementation instead of having to reimplement similar functionalities from scratch. The affine functions can be configured to have either integer or rational coefficients (parameter COEFFICIENTS in Figure 1).

7.3 Widening

The widening operator predicts the value of a piecewise-defined function over its still undefined partitions; this enforces convergence of the analysis of loops.

FuncTion can be configured (parameter HEURISTIC in Figure 1) to employ various widening heuristics [8]. The default heuristics does some sort of linear interpolation of functions with increasing gradients in adjacent partitions [34]; intuitively, the heuristic "prolongs" the gradient growth to the next adjacent partition on which the function is still undefined. Under the hood, the implementation relies on APRON for joining (i.e., computing the convex-hull of) the polyhedra representing the functions in adjacent partitions. The heuristic is particularly effective for programs looping over consecutive values of a variable.

Note that the widening prediction may not be correct and, if so, it needs to be repaired in order for the analysis to ultimately yield a correct result. For instance, the prediction might under-approximate the value of the concrete program semantics (cf. Section 3); an indication that this happened is when the value of the prediction increases in further iterations of the analysis [35]. In this case, the default behavior is to replace the prediction with the special value $\top$ to signal an unrecoverable loss of precision. Another widening heuristic consists instead in continuing the analysis with the incorrect prediction for a predefined number of iterations hoping that the value stabilizes. In our experience this simple heuristic turned out to be rather powerful [8]. We refer to [8, 35] for more details on the widening and all the available heuristics.

8 Tool Output

FuncTion returns TRUE if it can prove the chosen property for all possible inputs (that satisfy the given precondition, if any). Otherwise, it returns UNKNOWN. In addition, FuncTion returns a map from all program locations (i.e., CFG nodes) to the corresponding piecewise-defined functions. For each program location, the domain of the corresponding function is a sufficient precondition for the chosen property for all program executions from that program location. For liveness properties, the value of the function expresses (as a function of the program variables) an upper bound on the number of execution steps needed to reach the desired program state from that program location (for other properties the value remains 0 for all partitions of the domain of the function).

9 Example

Let us consider a simplified version of the loop in Figure 2a (which we have explicitly annotated with program control points):

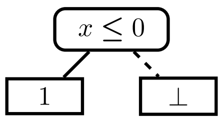
```
while $^{1}$( $x > 0$ ) {
    $^{2}x = x - y;$
}$^{3}$
```

We present here its conditional termination analysis performed by FuncTion. We instantiate the piecewise-defined functions abstract domain (cf. Section 7) with polyhedral constraints at the decision nodes (cf. Section 7.1) and affine functions at the leaf nodes (cf. Section 7.2).

The starting point of the analysis is the constant function equal to zero at the loop final control point **3** (cf. Section 6):

$$\boxed{0} \tag{1}$$

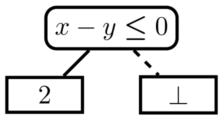
The function is then propagated backwards towards the loop initial control point 1 taking the (negation of the) loop condition into account and counting one step to termination (i.e., testing the loop condition):



(2)

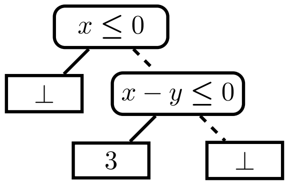
The inferred decision tree at this point indicates that the loop terminates in at most one program step if the loop condition is not satisfied (i.e., if $x \leq 0$).

The analysis proceeds to analyze the body of the loop taking into account the assignment $x = x - y$ at program control point **2**:



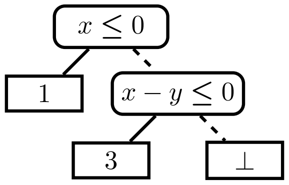
(3)

and the loop condition $x > 0$ towards program control point **1**:



(4)

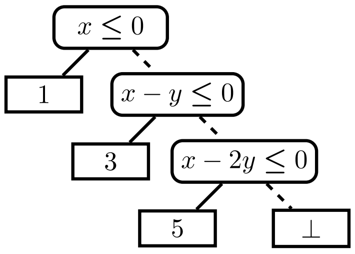
Joining the decision trees (2) and (4) yields the following decision tree at program control point **1**:



(5)

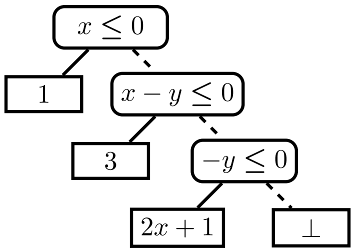
which indicates that the loop terminates in at most one program step if $x \leq 0$, and in at most three program steps if the loop is executed once (i.e., if $x \leq y$).

At the next iteration, we have the following at program control point **1**:



(6)

At this point, we need to perform a widening between the decision trees (5) and (6) to extrapolate a ranking function for the still undefined partitions (cf. Section 7.3). In this case, a very effective widening heuristic [8] inspired by Bagnara et al. [3] yields the following decision tree:



(7)

which indicates that the loop terminates in (at most) one program step if $x \leq 0$, in (at most) three steps if $x \leq y$, and in at most $2x + 1$ program steps if $y > 0$. Note that, $2x + 1$ does not depend on the value of y and so it is an over-approximation of the number of program steps actually needed for termination.

A further iteration of the analysis yields again the decision tree (7) which indicates that the analysis has converged and found the sufficient (and, in this case, also necessary) condition for termination of the loop $x \leq 0 \vee y > 0$.

10 Dead Ends

In the following, we discuss a couple of bad initial design decisions which were abandoned in favor of the current version of FuncTion.

Analysis on the AST. A first initial bad idea was performing the analysis directly on the program AST. This spared us from building its CFG and implementing a loop-detection algorithm. However, this also made the analysis engine unnecessarily complex, to support the full program abstract syntax. In particular, supporting function calls and `goto`s was very cumbersome. In the end, performing the analysis on the CFG allowed us to support a larger language syntax all the while considerably simplifying the core of FuncTion.

Internal Representation of Functions as Lists. The first version of FuncTion did not represent piecewise-defined functions using decision trees but rather as lists of function pieces. Each piece was a pair of numerical domain elements representing the domain and the value of the function piece, respectively. This made the implementation considerably simpler but critically compromised the performance of the tool. Investing in the current internal representation was the key idea to scale the analysis to larger programs.

11 Future Work

An extension of FuncTion [31] supports proving *robust (un)reachability* [22]. More generally, unless specified, FuncTion automatically infers which program variable values should be controlled (and under which constraints) for the property of interest to be satisfied.

In the future, we intend to make FuncTion's abstract domain bit-precise, e.g., by building upon existing bit-precise numerical abstract domain [32]. In addition, we plan to extend it to also support non-linear constraints, such as congruences [23], and non-linear functions, such as polynomials [6] or exponentials [18]. We would also like to investigate new widening heuristics.

Finally, other improvements are on the horizon, such as implementing a better loop-detection algorithm [26] and a better iteration strategy [5], supporting sparsity-preserving algorithms for octagonal constraints [21], and equipping FuncTion with a more precise interprocedural analysis [13].

Acknowledgements We are deeply grateful to Nathanaël Courant and Samuel Ueltschi and Antoine Miné for directly and indirectly contributing to the development of FuncTion.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21**(4), 181–185 (1985)
3. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise Widening Operators for Convex Polyhedra. Science of Computer Programming **58**(1-2), 28–56 (2005)
4. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static Analysis and Verification of Aerospace Software by Abstract Interpretation. In: AIAA, pp. 1–38 (2010)
5. Bourdoncle, F.: Efficient Chaotic Iteration Strategies With Widenings. In: FMPA, pp. 128–141 (1993)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination Analysis of Integer Linear Loops. In: CONCUR, pp. 488–502 (2005)
7. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In: Logic of Programs, pp. 52–71 (1981)
8. Courant, N., Urban, C.: Precise Widening Operators for Proving Termination by Abstract Interpretation. In: TACAS, pp. 136–152 (2017)
9. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Theoretical Computer Science **277**(1-2), 47–103 (2002)
10. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Symposium on Programming, pp. 106–130 (1976)
11. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL, pp. 238–252 (1977)
12. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL, pp. 269–282 (1979)
13. Cousot, P., Cousot, R.: Modular Static Program Analysis. pp. 159–178 (2002)
14. Cousot, P., Cousot, R.: An Abstract Interpretation Framework for Termination. In: POPL, pp. 245–258 (2012)
15. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, pp. 84–96 (1978)
16. Dimovski, A.S., Apel, S.: Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In: A. Møller, M. Sridharan (eds.) ECOOP, vol. 194, pp. 14:1–14:28 (2021)
17. D'Silva, V., Urban, C.: Conflict-Driven Conditional Termination. In: CAV, pp. 271–286 (2015)
18. Feret, J.: The Arithmetic-Geometric Progression Abstract Domain. In: VMCAI, pp. 42–58 (2005)
19. Floyd, R.W.: Assigning Meanings to Programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967)

20. Fuchs, H., Kedem, Z.M., Naylor, B.F.: On Visible Surface Generation by a Priori Tree Structures. SIGGRAPH Computer Graphics **14**(3), 124–133 (1980)
21. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting Sparsity in Difference-Bound Matrices. In: SAS, pp. 189–211 (2016)
22. Girol, G., Farinier, B., Bardin, S.: Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In: CAV, pp. 669–693 (2021)
23. Granger, P.: Static Analysis of Arithmetic Congruences. International Journal of Computer Math pp. 165–199 (1989)
24. Jeannet, B.: Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes. In: SAS, pp. 52–68 (2002)
25. Jeannet, B., Miné, A.: APRON: A Library of Numerical Abstract Domains for Static Analysis. In: CAV, p. 661––667 (2009)
26. Lengauer, T., Tarjan, R.E.: A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Trans. Program. Lang. Syst. **1**(1), 121–141 (1979)
27. Manna, Z., Pnueli, A.: The Temporal Verification of Reactive Systems: Progress (1996)
28. Miné, A.: The Octagon Abstract Domain. Higher Order and Symbolic Computation **19**(1), 31–100 (2006)
29. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
30. Pnueli, A.: The Temporal Logic of Programs. In: FOCS, pp. 46–57 (1977)
31. Remil, N.M., Urban, C., Miné, A.: Automatic Detection of Vulnerable Variables for CTL Properties of Programs. In: N.S. Bjørner, M. Heule, A. Voronkov (eds.) LPAR, *EPiC Series in Computing*, vol. 100, pp. 116–126. EasyChair (2024). doi:10.29007/DNPX. `https://doi.org/10.29007/dnpx`
32. Sharma, T., Reps, T.W.: Sound Bit-Precise Numerical Domains. In: VMCAI, pp. 500–520 (2017)
33. Turing, A.: Checking a Large Routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69 (1949)
34. Urban, C.: The Abstract Domain of Segmented Ranking Functions. In: SAS, pp. 43–62 (2013)
35. Urban, C.: Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. Ph.D. thesis, École Normale Supérieure (2015)
36. Urban, C., Miné, A.: A Decision Tree Abstract Domain for Proving Conditional Termination. In: SAS, pp. 302–318 (2014)
37. Urban, C., Miné, A.: An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In: ESOP, pp. 412–431 (2014)
38. Urban, C., Miné, A.: Inference of Ranking Functions for Proving Temporal Properties by Abstract Interpretation. Computer Languages, Systems & Structures **47**, 77–103 (2017)
39. Urban, C., Ueltschi, S., Müller, P.: Abstract Interpretation of CTL Properties. In: SAS, pp. 402–422 (2018)