Precise Widening Operators for Proving Termination by Abstract Interpretation

Nathanaëlle Courant[1] and Caterina Urban[2]

[1] École Normale Supérieure, France
nathanael.courant@ens.fr
[2] ETH Zurich, Switzerland
caterina.urban@inf.ethz.ch

Abstract FuncTion is a static analyzer designed for proving conditional termination of C programs by means of abstract interpretation. Its underlying abstract domain is based on piecewise-defined functions, which provide an upper bound on the number of program execution steps until termination as a function of the program variables.
In this paper, we fully parameterize various aspects of the abstract domain, gaining a flexible balance between the precision and the cost of the analysis. We propose heuristics to improve the fixpoint extrapolation strategy (i.e., the widening operator) of the abstract domain. In particular we identify new widening operators, which combine these heuristics to dramatically increase the precision of the analysis while offering good cost compromises. We also introduce a more precise, albeit costly, variable assignment operator and the support for choosing between integer and rational values for the piecewise-defined functions.
We combined these improvements to obtain an implementation of the abstract domain which subsumes the previous implementation. We provide experimental evidence in comparison with state-of-the-art tools showing a considerable improvement in precision at a minor cost in performance.

1 Introduction

Programming errors which cause non-termination can compromise software systems by making them irresponsive. Notorious examples are the Microsoft Zune Z2K bug[3] and the Microsoft Azure Storage service interruption[4]. Termination bugs can also be exploited in denial-of-service attacks[5].Therefore, proving program termination is important for ensuring software reliability.

The traditional method for proving termination is based on the synthesis of a *ranking function*, a well-founded metric which strictly decreases during the program execution. FuncTion [36] is a static analyzer which automatically infers ranking functions and sufficient precondition for program termination by means

[3] http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/
[4] http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption/
[5] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1890

of abstract interpretation [13]. The tool is based on the abstract interpretation framework for termination introduced by Cousot and Cousot [14]

The underlying abstract domain of FuncTion is based on *piecewise-defined ranking functions* [40], which provide an upper bound on the number of program execution steps until termination as a function of the program variables. The piecewise-defined functions are represented by *decision trees*, where the decision nodes are labeled by linear constraints over the program variables, and the leaf nodes are labeled by functions of the program variables.

In this paper, we fully parameterize various aspects of the abstract domain, gaining a flexible balance between the precision and the cost of the analysis. We propose options to tune the representation of the domain and value of the ranking functions manipulated by the abstract domain. In particular, we introduce the support for using rational coefficients for the functions labeling the leaf nodes of the decision trees, all the while strengthening their decrease condition to still ensure termination. We also introduce a variable assignment operator which is very effective for programs with unbounded non-determinism. Finally, we propose heuristics to improve the widening operator of the abstract domain. Specifically, we suggest an heuristic inspired by [1] to infer new linear constraints to add to a decision tree and two heuristics to infer a value for the leaf nodes on which the ranking function is not yet defined. We identify new widening operators, which combine these heuristics to dramatically increase the precision of the analysis while offering good cost compromises.

We combined these improvements to obtain an implementation of the abstract domain which subsumes the previous implementation. We provide experimental evidence in comparison with state-of-the-art tools [34,21,22] showing a considerable improvement in precision at a minor cost in performance.

Outline. Section 2 offers a glimpse into the theory behind proving termination by abstract interpretation. In Section 3, we recall the ranking functions abstract domain and we discuss options to tune the representation of the piecewise-defined functions manipulated by the abstract domain. We suggest new precise widening operators in Section 4. Section 5 presents the result of our experimental evaluation. We discuss related work in Section 6 and Section 7 concludes.

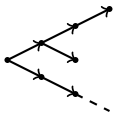
2 Termination and Ranking Functions

The traditional method for proving program termination dates back to Turing [35] and Floyd [17]. It consists in inferring a *ranking function*, namely a function from the program states to elements of a well-ordered set whose value decreases during program execution. The best known well-ordered sets are the natural numbers $\langle \mathbb{N}, \leq \rangle$ and the ordinals $\langle \mathbb{O}, \leq \rangle$, and the most obvious ranking function maps each program state to the number of program execution steps until termination, or some well-chosen upper bound on this number.

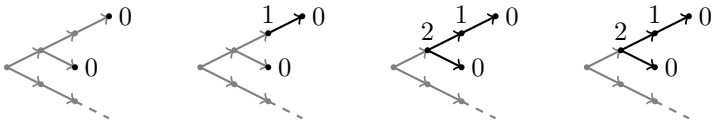
In [14], Cousot and Cousot formalize the notion of a most precise ranking function w for a program. Intuitively, it is a partial function defined starting

from the program final states, where it has value zero, and retracing the program backwards while mapping each program state definitely leading to a final state (i.e., a program state such that all program execution traces to which it belongs are terminating) to an ordinal representing an upper bound on the number of program execution steps remaining to termination. The domain $\mathrm{dom}(w)$ of w is the set of states from which the program execution must terminate: all traces branching from a state $s \in \mathrm{dom}(w)$ terminate in at most $w(s)$ execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(w)$ does not terminate.

Example 1. Let us consider the following execution traces of a given program:



The most precise ranking function for the program is iteratively defined as:



where unlabelled states are outside the domain of the function.

The most precise ranking function w is sound and complete to prove program termination [14]. However, it is usually not computable. In the following sections we recall and present various improvements on decidable approximations of w [40]. These *over-approximate* the value of w and *under-approximate* its domain of definition $\mathrm{dom}(w)$. In this way, we infer sufficient preconditions for program termination: if the approximation is defined on a program state, then all execution traces branching from that state are terminating.

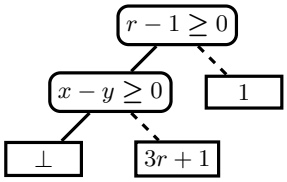
3 The Ranking Functions Abstract Domain

We use abstract interpretation [13] to approximate the most precise ranking function mentioned in the previous section. In [40], to this end, we introduce an abstract domain based on *piecewise-defined ranking functions*. We recall here (and in the next section) the features of the abstract domain that are relevant for our purposes and introduce various improvements and parameterizations to tune the precision of the abstract domain. We refer to [37] for an exhaustive presentation of the original ranking functions abstract domain.

The elements of the abstract domain are piecewise-defined partial functions. Their internal representation is inspired by the space partitioning trees [18] developed in the context of 3D computer graphics and the use of decision trees in program analysis and verification [3,24]: the piecewise-defined partial functions are represented by *decision trees*, where the decision nodes are labeled by linear constraints over the program variables, and the leaf nodes are labeled

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

(a)



(b)

Figure 1: Decision tree representation (b) of the piecewise-defined ranking function for a simple C function (a). The linear constraints 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 value $\bot$ explicitly represents the undefined partition of the partial function.

by functions of the program variables. The decision nodes recursively partition the space of possible values of the program variables and the functions at the leaves provide the corresponding upper bounds on the number of program execution steps until termination. An example of decision tree representation of a piecewise-defined ranking function is shown in Figure 1.

The partitioning is dynamic: during the analysis, partitions (resp. decision nodes and constraints) are split (resp. added) by tests, modified by variable assignments and joined (resp. removed) when merging control flows. In order to minimize the cost of the analysis, a widening limits the height of the decision trees and the number of maintained partitions.

The abstract domain is parameterized in various aspects. Figure 2 offers an overview of the various parameterizations currently available. We discuss here options to tune the representation of the domain and value of the ranking functions manipulated by the abstract domain. The discussion on options to tune the precision of the widening operator is postponed to the next section.

3.1 Domain Representation

The domain of a ranking function represented by a decision tree is partitioned into pieces which are determined by the *linear constraints* encountered along the paths to the leaves of the tree. The abstract domain supports linear constraints of different expressivity. In the following, we also propose an alternative strategy to modify the linear constraints as a result of a variable assignment. We plan to support non-linear constraints as part of our future work.

Linear Constraints. We rely on existing numerical abstract domains for labeling the decision nodes with the corresponding linear constraints and for manipulating them. In order of expressivity, we support interval [12] constraints (i.e., of the form $\pm x \leq c$), octagonal [30] constraints (i.e., of the form $\pm x_i \pm x_j \leq c$), and

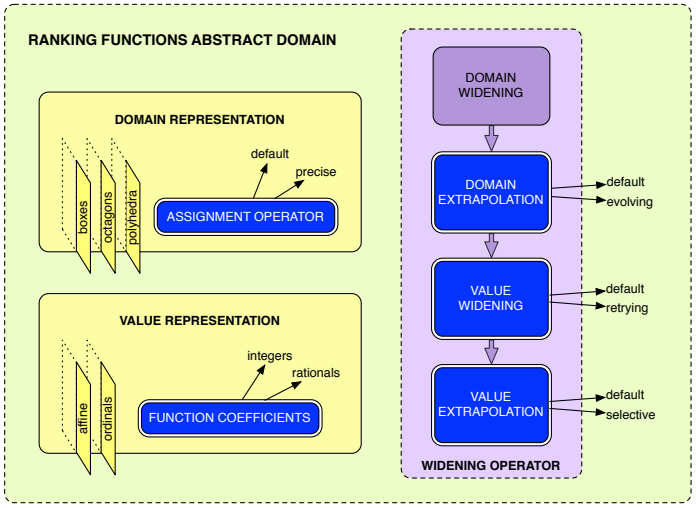


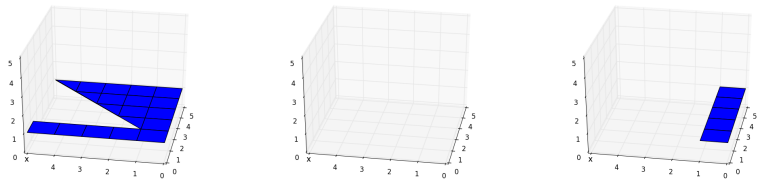
Figure 2: Overview of the various parameterizations for the components of the ranking function abstract domain. Highlighted with a double border are the components for which new parameterizations are introduced in this paper.

polyhedral [15] constraints (i.e., of the form $c_1 \cdot x_1 + \cdots + c_k \cdot x_k \leq c_{k+1}$). As for efficiency, contrary to expectations, octagonal constraints are the costliest labeling in practice. The reason for this lies in how constraints are manipulated as a results of a variable assignment which amplifies known performance drawbacks for octagons [19,26]. We expand on this shortly.

Assignment Operator. A variable assignment might impact some of the linear constraints within the decision nodes as well as some functions within the leaf nodes. The abstract domain now supports two strategies to modify the decision trees as a result of a variable assignment:

- The default strategy [40] consists in carrying out a variable assignment independently on each linear constraint labeling a decision node and each function labeling a leaf of the decision tree. This strategy is cheap since it requires a single tree traversal. It is sometimes imprecise as shown in Figure 3.
- The new precise strategy consists in carrying out a variable assignment on each partition of a ranking function and then merging the resulting partitions. This strategy is costlier since it requires traversing the initial decision tree to identify the initial 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 linear constraints possibly higher than the height of the initial decision tree [37]. However, this strategy is much more precise as shown in Figure 3.

Both strategies do not work well with octagonal constraints. It is known that the original algorithms for manipulating octagons do not preserve their sparsity



(a) (b) (c)

Figure 3: Impact of the non-deterministic variable assignment `x = ?` (which resets the value of `x` to a randomly chosen value) on a ranking function (a) using the default assignment strategy (b) and the precise assignment strategy (c). Note that the default assignment strategy loses all information.

[19,26]. An immediate consequence of this is that a variable assignment on a single octagonal constraints often yields multiple linear constraints. This effect is particularly amplified by the default assignment strategy described above. The precise assignment strategy suffers less from this but the decision trees still tend to grow considerably in size. We plan to support sparsity-preserving algorithms for octagonal constraints as part of our future work.

3.2 Value Representation

The functions used for labeling the leaves of the decision trees are *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 $\bot$ and $\top$ which explicitly represent undefined functions (cf. Figure 1b). The element $\top$ shares the same meaning of $\bot$ but is only introduced by the widening operator. We expand on this in the next section. More specifically, we support *lexicographic* affine *functions* $(f_k, \ldots, f_1, f_0)$ in the isomorphic form of ordinals $\omega^k \cdot f_k + \cdots + \omega \cdot f_1 + f_0$ [29,39]. The *maximum degree* k of the polynomial is a parameter of the analysis. We leave non-linear functions for future work.

The coefficients of the affine functions are by default integers [40] and we now also support rational coefficients. Note that, when using rational coefficients, the functions have to decrease by at least one at each program execution step to ensure termination. Indeed, a decreasing sequence of rational number is not necessarily finite. However, the integer parts of rational-valued functions which decrease by at least one at each program step yield a finite decreasing sequence.

4 The Widening Operator on Ranking Functions

The widening operator $\triangledown$ tries to predict a value for the ranking function over the states on which it is not yet defined. Thus, it has more freedom than traditional widening operators, in the sense that it is temporarily allowed to *under-approximate* the value of the most precise ranking function w (cf. Section 2) or

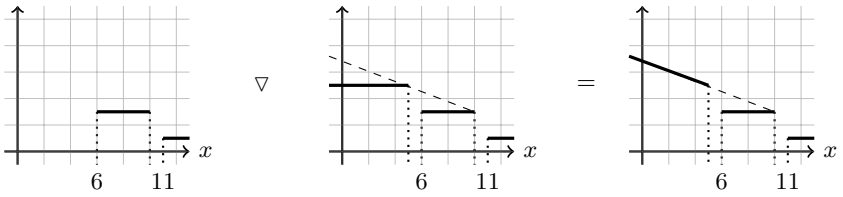


Figure 4: Example of *value extrapolation*. The default heuristics increases the slope of the function defined for $x < 6$ with respect to the value of the function defined in its adjacent partition (i.e., for $6 \leq x < 11$).

over-approximate its domain of definition $\mathrm{dom}(w)$, or both — in contrast with the observation made at the end of Section 2. The only requirement is that these discrepancies are resolved before the analysis stabilizes.

In more detail, give two decision trees t_1 and t_2, the widening operator will go through the following steps to compute $t_1 \triangledown t_2$ [40]:

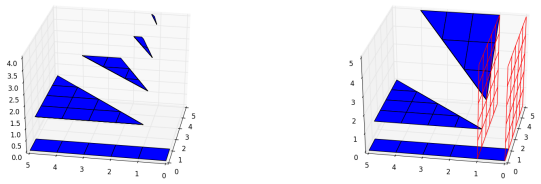
Domain Widening. This step resolves an eventual over-approximation of the domain $\mathrm{dom}(w)$ of w following the inclusion of a program state from which a non-terminating program execution is reachable. This discrepancy manifests itself when a leaf in t_1 is labeled by a function and its corresponding leaf in t_2 is labeled by $\bot$. The widening operator marks the offending leaf in t_2 with $\top$ to prevent successive iterates of the analysis from mistakenly including again the same program state into the domain of the ranking function.

Domain Extrapolation. This step extrapolates the domain of the ranking functions over the states on which it is not yet defined. The default strategy consists in dropping the decision nodes that belong to t_2 but not to t_1 and merging the corresponding subtrees[6]. In this way we might lose information but we ensure convergence by limiting the size of the decision trees.

Value Widening. This step resolves an eventual under-approximation of the value of w and an eventual over-approximation of the domain $\mathrm{dom}(w)$ of w following the inclusion of a non-terminating program state. These discrepancies manifest themselves when the value of a function labeling a leaf in t_1 is smaller than the value of the function labeling the corresponding leaf in t_2. In this case, the default strategy consists again in marking the offending leaf in t_2 with $\top$ to exclude it from the rest of the analysis.

Value Extrapolation. This step extrapolates the value of the ranking function over the states that have been added to the domain of the ranking function in the last analysis iterate. These states are represented by the leaves in t_2 that are labeled by a function and their corresponding leaves in t_1 are labeled by $\bot$. The default heuristic consists in increasing the gradient of the functions

[6] We requires the decision nodes belonging to t_1 to be a subset of those belonging to t_2. This can always be ensured by computing $t_1 \triangledown (t_1 \sqcup t_2)$ instead of $t_1 \triangledown t_2$.



(a) (b)

Figure 5: The ranking function (b) obtained after widening using the evolving strategy on a given ranking function (a). Highlighted in red are the linear constraints inferred by the strategy, which limit the domain extrapolation to the increasingly smaller pieces on which the given ranking function is defined.

with respect to the functions labeling their adjacent leaves in the decision tree. The rationale being that programs often loop over consecutive values of a variable, we use the information available in adjacent partitions of the domain of the ranking function to infer the shape of the ranking function for the current partitions. An example is shown in Figure 4.

In the rest of the section, we suggest new heuristics to improve the default strategies used in the last three steps performed by the widening operator. These yield new widening operators which combine these heuristics to dramatically increase the precision of the analysis while offering good cost compromises.

Note that, to improve precision, it is customary to avoid the use of the widening operator for a certain number of analysis iterates. In the following, we refer to this number as *delay threshold*.

4.1 Domain Extrapolation

The default strategy for the *domain extrapolation* never infers new linear constraints and this hinders proving termination for some programs. In the following, we propose an alternative strategy which limits the number of decision nodes to be dropped during the analysis and labels them with new linear constraints. It is important to carefully choose the new added constraints to avoid slowing down the analysis unnecessarily and to make sure that the analysis still converges.

We suggest here a strategy inspired by the *evolving rays* heuristic presented in [1] to improve the widening operator of the polyhedra abstract domain [15]. The evolving strategy examines each linear constraint c_2 in t_2 (i.e., the decision tree corresponding to the last iterate of the analysis) as if it was generated by *rotation* of a linear constraint c_1 in t_1 (i.e., the decision tree corresponding to

the previous iterate of the analysis). This rotation is formalized as follows [1]:

$$\mathsf{evolve}(u, v) = w$$
$$\text{where } w_i = \begin{cases} 0 & \text{if } \exists j \in \{1, \ldots, n\}.(u_i v_j - u_j v_i) u_i u_j < 0 \\ u_i & \text{otherwise} \end{cases}$$

where u and w are the vectors of coefficients of the linear constraints c_2 in t_2 and c_1 in t_1, respectively. In particular, evolve sets to zero the components of u that match the direction of rotation. Intuitively, the evolving strategy continues the rotation of c_2 until one or more of the non-null coefficients of c_2 become zero. The new constraint reaches one of the boundaries of the orthant where c_2 lies without trespassing it. This strategy is particularly useful in situations similar to the one depicted in Figure 5a: the ranking function is defined over increasingly smaller pieces delimited by different rotations of a linear constraint. In such case, the evolving strategy infers the linear constraints highlighted in red in Figure 5b, thus extrapolating the domain of the ranking function up to the boundary of the orthant where the function is defined.

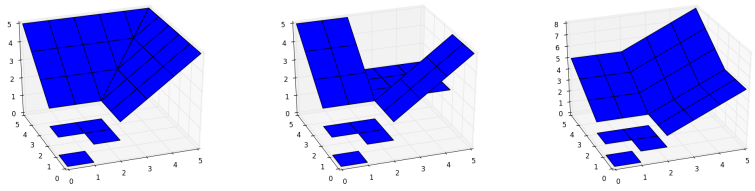
More specifically, the evolving strategy explores each pair of linear constraints on the same path in the decision tree t_2 and modifies them as described above to obtain new constraints. The strategy then discards the less frequently obtained constraints. The relevant frequency is a parameter of the analysis which in the following we call the *evolving threshold*. In our experience, it is usually a good choice to set the evolving threshold to be equal to the delay threshold of the widening. The remaining constraints are used to substitute the linear constraints that appear in t_2 but not in t_1, possibly merging the corresponding subtrees.

Note that, by definition, the number of new linear constraints that can be added by the evolving strategy is finite. The strategy then defaults to the default strategy and this guarantees the termination of the analysis.

4.2 Value Widening

The default strategy for the *value widening* marks with $\top$ the leaves in t_2 (i.e., the decision tree corresponding to the last iterate of the analysis) labeled with a larger value than their corresponding leaves in t_1 (i.e., the decision tree corresponding to the previous iterate of the analysis). This resolves eventual discrepancies in the approximation of the most precise ranking function w at the cost of losing precision in the analysis. As an example, consider the situation shown in Figure 6: Figure 6a depicts the most precise ranking function for a program and Figure 6b depicts its approximation at the iterate immediately after widening. Note that one partition of the ranking function shown in Figure 6b under-approximates the value of the ranking function shown in Figure 6a. The default strategy would then label the offending partition with $\top$, in essence giving up on trying to predict a value for the ranking function on that partition.

A simple and yet powerful improvement is to maintain the values of the offending leaves in t_2 and continue the analysis. In this way, the analysis can



(a) (b) (c)

Figure 6: The ranking function (c) obtained after widening using the retrying strategy on a given ranking function (b). Note that the given ranking function (b) under-approximates the value of the ranking function shown in (a).

do various attempts at predicting a stable value for the ranking function. Note that using this retrying strategy without caution would cause the analysis to not converge for a number of programs. Instead, we limit the number of attempts to a certain *retrying threshold*, and then revert to the default strategy.

The retrying strategy for ordinals of the form $\omega^k \cdot f_k + \cdots + \omega \cdot f_1 + f_0$ (cf. Section 3.2) behaves analogously to the other abstract domain operators for manipulating ordinals [39]. It works in ascending powers of ω carrying to the next higher degree when the *retrying threshold* has been reached (up to the maximum degree for the polynomial, in which case we default to $\top$).

4.3 Value Extrapolation

The default heuristic for the *value extrapolation* consists in increasing the gradient of the ranking function with respect to its value in *adjacent* partition of its domain. Note that, many other heuristics are possible. In fact, this step only affects the precision of the analysis, and not its convergence or its soundness.

In this paper, we propose a selective extrapolation heuristic, which increases the gradient of the ranking function with respect to *selected* partitions of its domain. More specifically, the heuristic selects the partitions from which the current partition is reachable in one loop iteration. This strategy is particularly effective in combination with the evolving strategy described in Section 4.1. Indeed, the evolving strategy often splits partitions by adding new linear constraints and, in some cases, this affects the precision of the analysis since it alters the adjacency relationships between the pieces on which the ranking function is defined.

We plan to investigate other strategies as part of our future work.

5 Implementation and Experimental Evaluation

The ranking functions abstract domain and the new parameterizations introduced in this paper are implemented in FuncTion [36] and are available online[7]. The implementation is in OCaml and consists of around 3K lines of code. The current front-end of FuncTion accepts programs written in a (subset of) C, without `struct` and `union` types. It provides only a limited support for arrays, pointers, and recursion. The only basic data type are mathematical integers, deviating from the standard semantics of C. The abstract domain builds on the numerical abstract domains provided by the APRON library [25].

The analysis proceeds by structural induction on the program syntax, iterating loops until a fixpoint is reached. In case of nested loops, a fixpoint on the inner loop is computed for each iteration of the outer loop, following [4,31]. It is also possible to refine the analysis by only considering the reachable states.

Experimental Evaluation. The ranking functions abstract domain was evaluated on 242 terminating C programs collected from the *5th International Competition on Software Verification (SV-COMP 2016)*. Due to the limitations in the current front-end of FuncTion we were not able to analyze 47% of the test cases. The experiments were performed on a system with a 3.20GHz 64-bit Dual-Core CPU (Intel i5-3470) and 6GB of RAM, running Ubuntu 16.04.1 LTS.

We compared multiple configurations of parameters for the abstract domain. We report here the result obtained with the most relevant configurations. Unless otherwise specified, the common configuration of parameters uses the default strategy for handling variable assignments (cf. Section 3.1), a *maximum degree* of two for ordinals using integer coefficients for affine functions (cf. Section 3.2), and a *delay threshold* of three for the widening (cf. Section 4). Figure 7 presents the results obtained using polyhedral constraints. Figure 8 shows the successful configurations for each test case. Using interval constraints yields fewer successful test cases (around 50% less successes) but it generally ensures better runtimes. The exception is a slight slowdown of the analysis when using rational coefficients, which is not observed when using polyhedral constraints. We did not evaluate the use of octagonal constraints due to the performance drawbacks discussed in Section 3.1. We used a time limit of 300 seconds for each test case.

We can observe that using the retrying strategy always improves the overall analysis result: configurations 3, 6, 7, and 9 are more successful than the corresponding configurations 1, 4, 5, and 8, which instead use the default strategy. In particular, configuration 3 is the best configuration in terms of number of successes (cf. Figure 7). However, in general, improving the precision of the widening operator does not necessarily improve the overall analysis result. More specifically, configurations 4 to 9 seem to perform generally worse than configuration 1 and 3 both in terms of number of successes and running times. However, although these configurations are not effective for a number of programs for which configuration 1 and 3 are successful, they are not subsumed by them since they allow proving termination of many other programs (cf. Figure 8).

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

N	Value		Widening				Assignment		Success	Time	TO
	$\mathbb{N}$	$\mathbb{Q}$	delay	retrying	evolving	selective	default	precise			
1	✓						✓		128	0.35s	5
									140	0.44s	3
2		✓					✓		124	0.35s	5
									138	0.78s	3
3	✓			✓			✓		138	0.40s	6
									152	0.48s	3
4	✓				✓		✓		125	1.28s	11
									127	1.01s	10
5	✓					✓	✓		118	0.28s	5
									106	0.21s	3
6	✓			✓	✓		✓		136	1.74s	17
									139	1.18s	14
7	✓			✓		✓	✓		129	0.35s	5
									124	0.26s	4
8	✓				✓	✓	✓		116	0.92s	11
									102	0.31s	10
9	✓			✓	✓	✓	✓		134	1.40s	16
									123	0.63s	19
10		✓		✓	✓	✓	✓		128	1.41s	18
									120	0.48s	16
11	✓			✓				✓	133	3.41s	50
									132	6.59s	56
12	✓		6	✓				✓	122	8.70s	92
									120	6.22s	98

Figure 7: Evaluation of FuncTion using polyhedral constraints. For each configuration N, the bottom row corresponds to the results obtained by restricting the analysis to the reachable states, and the top row to the results obtained without reachability information. Highlighted in blue is the best configuration in terms of number of successes. Time is the mean time per successful test case.

Another interesting observation is that using rational coefficients in configuration 2 worsens the result of the analysis compared to configuration 1 which uses integer coefficients (cf. Figure 8). Instead, using rational coefficients in configuration 10 allows proving termination for a number of programs for which configuration 9 (which uses integer coefficients) is unsuccessful.

The configurations using the evolving strategy (i.e., 4, 6, 8, 9, and 10) tend to be slower than the configurations which use the default strategy. As a consequence, they suffer from a higher number of timeouts (cf. Figure 7). Even worse is the slowdown caused by the precise strategy to handle variable assignments (cf. configurations 11 and 12) and a higher *delay threshold* for the widening (cf. configuration 12). We observed that a *delay threshold* higher than six only marginally improves precision while significantly worsening running times.

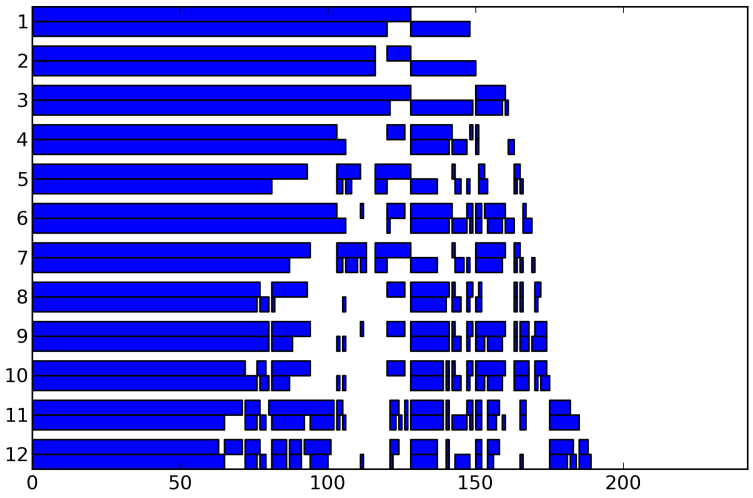


Figure 8: Test case coverage for the evaluation of FuncTion using polyhedral constraints. The horizontal axis enumerates the test cases. For each configuration of FuncTion given on the vertical axis (without and with reachability information, as in Figure 7), a colored area corresponds to successful test cases.

Finally, we observed that there are some configurations for which decreasing the precision of the linear constraints (from polyhedral to interval constraints) allows proving termination of some more programs. In particular, this concerns configuration 2 as well as some of the other configurations when limiting the analysis to the reachable states. However, this happens very rarely: overall, only three programs can be proven terminating only using interval constraints.

We also compared FuncTion against the tools participating to *SV-COMP 2016*: AProVE [34], SeaHorn [21,38] and UAutomizer [22]. We did not compare with other tools such as T2 [6] and 2LS [9] since FuncTion does not yet support the input format of T2 and bit-precise integer semantics (like 2LS does). As we observed that most of the parameter configurations of the abstract domain do not subsume each other, for the comparison, we set up FuncTion to use multiple parameter combinations successively, each with a time limit of 25 seconds. More specifically, we first use configuration 3, which offers the best compromise between number of successes and running times. We then move onto configurations that use the evolving strategy and the selective strategy, which are successful for other programs at the cost of an increased running time. Finally, we try the even more costly configurations that use the precise strategy for handling variable assignments and a higher *delay threshold* for the widening.

We ran FuncTion on the same machine as above, while for the other tools we used the results of *SV-COMP 2016* since our machine was not powerful enough

	■	▲	×	○	Success	Time	TO
FuncTion	–	–	–	–	195	5.25s	7
AProVE [34]	7	36	188	11	224	15.66s	15
SeaHorn [21,38]	31	22	164	25	186	8.57s	52
UAutomizer [22]	10	36	185	11	221	14.04s	6

Figure 9: Comparison of FuncTion against tools participating in SV-COMP 2016. ■ denotes the number of programs for which only FuncTion was successful, ▲ the number of programs for which only the other tool was successful, × the number for which both tools were successful and ○ the number for which neither tool was. Time corresponds to the mean time per success of the tool.

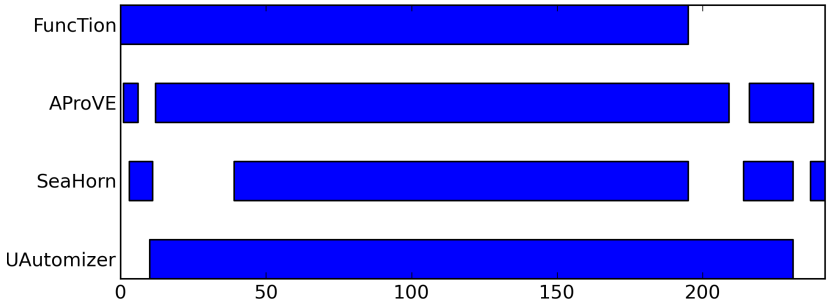


Figure 10: Test case coverage for the comparison of FuncTion against tools participating in SV-COMP 2016. The horizontal axis enumerates the test cases. Each colored area corresponds to successful test cases.

to run them. The time limit per test case was again 300 seconds. Figure 9 shows the result of the comparison and Figure 10 shows the successful tools for each test case. We can observe that, despite being less successful than AProVE or UAutomizer, FuncTion is able to prove termination of an important number of programs (i.e., 80% of the test cases, cf. Figure 9). Moreover, FuncTion is generally faster than all other tools, despite the fact that these were run on more powerful machines. Finally, we can observe in Figure 10, that for each tool there is a small subset of the test cases for which it is the only successful tool. The four tools together are able to prove termination for all the test cases.

6 Related Work

In the recent past, termination analysis has benefited from many research advances and powerful termination provers have emerged over the years.

AProVE [34] is probably the most mature tool in the field. Its underlying theory is the size-change termination approach [27], originated in the context of term rewriting systems, which consists in collecting a set of size-change graphs

(representing function calls) and combining them into multipaths (representing program executions) in such a way that at least one variable is guaranteed to decrease. Compared to size-change termination, FuncTion avoids the exploration of the combinatorial space of multipaths by manipulating ordinals.

Terminator [10] is based on the transition invariants method introduced in [33]. More specifically, the tool iteratively constructs transition invariants by searching within a program for single paths representing potential counterexamples to termination, computing a ranking function for each one of them individually (as in [32]), and combining the obtained ranking functions into a single termination argument. Its successor, T2 [6], has abandoned the transition invariants approach in favor of lexicographic ranking functions [11] and has broadened its scope to a wide range of temporal properties [7].

UAutomizer [22] is a software model checker based on an automata-theoretic approach to software verification [23]. Similarly to Terminator, it reduces proving termination to proving that no program state is repeatedly visited (and it is not covered by the current termination argument), and composes termination arguments by repeatedly invoking a ranking function synthesis tool [28]. In contrast, the approach recently implemented in the software model checker SeaHorn [21] systematically samples terminating program executions and extrapolates from these a ranking function [38] using an approach which resembles the *value extrapolation* of the widening operator implemented in FuncTion.

Finally, another recent addition to the family of termination provers is 2LS [9], which implements a *bit-precise* inter-procedural termination analysis. The analysis solves a series of second-order logic formulae by reducing them to first-order using polyhedral templates. In contrast with the tools mentioned above, both 2LS and FuncTion prove *conditional* termination.

7 Conclusion and Future Work

In this paper, we fully parameterized various aspects of the ranking function abstract domain implemented in the static analyzer FuncTion. We identified new widening operators, which increase the precision of the analysis while offering good cost compromises. We also introduced options to tune the representation of the ranking functions manipulated by the abstract domain. In combining these improvements, we obtained an implementation which subsumes the previous implementation and is competitive with state-of-the-art termination provers.

In the future, we would like to extend the abstract domain to also support non-linear constraints, such as congruences [20], and non-linear functions, such as polynomials [5] or exponentials [16]. In addition, we plan to support sparsity-preserving algorithms for manipulating octagonal constraints [19,26]. We would also like to investigate new strategies to predict a value for the ranking function during widening. Finally, we plan to work on proving termination of more complex programs, such as heap-manipulating programs. We would like to investigate the adaptability of existing methods [2] and existing abstract domains for heap analysis [8], and possibly design new techniques.

References

1. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.
2. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV*, pages 386–400, 2006.
3. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. In *AIAA*, 2010.
4. F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *FMPA*, pages 128–141, 1993.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. The Polyranking Principle. In *ICALP*, pages 1349–1361, 2005.
6. M. Brockschmidt, B. Cook, and C. Fuhs. Better Termination Proving through Cooperation. In *CAV*, pages 413–429, 2013.
7. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal Property Verification. In *TACAS*, pages 387–393, 2016.
8. B. E. Chang and X. Rival. Modular construction of shape-numeric analyzers. In *Festschrift for Dave Schmidt*, pages 161–185, 2013.
9. H. Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising Interprocedural Bit-Precise Termination Proofs. In *ASE*, pages 53–64, 2015.
10. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In *CAV*, pages 415–418, 2006.
11. B. Cook, A. See, and F. Zuleger. Ramsey vs. Lexicographic Termination Proving. In *TACAS*, pages 47–61, 2013.
12. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.
13. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
14. P. Cousot and R. Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.
15. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
16. J. Feret. The Arithmetic-Geometric Progression Abstract Domain. In *VMCAI*, pages 42–58, 2005.
17. R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
18. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a Priori Tree Structures. *SIGGRAPH Computer Graphics*, 14(3):124–133, 1980.
19. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting Sparsity in Difference-Bound Matrices. In *SAS*, pages 189–211, 2016.
20. P. Granger. Static Analysis of Arithmetic Congruences. *International Journal of Computer Math*, pages 165–199, 1989.
21. A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A Framework for Verifying C Programs (Competition Contribution). In *TACAS*, pages 447–450, 2015.
22. M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski. Ultimate Automizer with Two-Track Proofs - (Competition Contribution). In *TACAS*, pages 950–953, 2016.

23. M. Heizmann, J. Hoenicke, and A. Podelski. Software Model Checking for People Who Love Automata. In *CAV*, pages 36–52, 2013.
24. B. Jeannet. Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes. In *SAS*, pages 52–68, 2002.
25. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.
26. J.-H. Jourdan. Sparsity Preserving Algorithms for Octagons. In *NSAD*, 2016.
27. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL*, pages 81–92, 2001.
28. J. Leike and M. Heizmann. Ranking Templates for Linear Loops. In *TACAS*, pages 172–186, 2014.
29. Z. Manna and A. Pnueli. The Temporal Verification of Reactive Systems: Progress, 1996.
30. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
31. K. Muthukumar and M. V. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, 1992.
32. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.
33. A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.
34. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *TACAS*, pages 417–419, 2015.
35. A. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
36. C. Urban. FuncTion: An Abstract Domain Functor for Termination (Competition Contribution). In *TACAS*, pages 464–466, 2015.
37. C. Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs*. PhD thesis, École Normale Supérieure, July 2015.
38. C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing Ranking Functions from Bits and Pieces. In *TACAS*, 2016.
39. C. Urban and A. Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *ESOP*, pages 412–431, 2014.
40. C. Urban and A. Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In *SAS*, pages 302–318, 2014.