Rapport de stage : Precise Widenings for Proving Termination by Abstract Interpretation

Nathanaël Courant, sous la direction de Caterina Urban

Juin-Juillet 2016

Abstract

Ce rapport décrit les résultats obtenus au cours de mon stage de fin de licence 3 à l'ETH Zürich, sous la direction de Caterina Urban.

Le but du stage était d'étudier l'opérateur de widening de FuncTion, un programme permettant de montrer la terminaison de programmes C, de déterminer dans quels cas et pourquoi il perdait en précision, puis de chercher des heuristiques pour remédier à cela. Une troisième étape consistait alors à implanter ces heuristiques et les évaluer expérimentalement.

Le stage s'est déroulé dans les locaux de l'ETH Zürich, en compagnie d'autres membres de l'équipe et d'autres stagiaires. Cela m'a permis d'échanger avec un certain nombre de chercheurs, et de découvrir ce que faisait l'équipe.

La première section de ce rapport décrit rapidement l'opérateur de widening de FuncTion existant ; les suivantes présentent mon propre travail au cours de ce stage.

Contents

1 Notations **2**

2 Overview of FuncTion's Widening Operator **2**
2.1 Introduction 2
2.2 The Decision Trees Abstract Domain 2
2.3 The Widening Operator on Decision Trees 3

3 Improving the Widening Operator **4**
3.1 Retrying when Prediction was Incorrect 4
3.2 Evolving Rays 6
3.3 Extending Towards Relevant Segments 8
3.4 Putting it all together 9
3.5 Other Improvements 10

4 Experimental Evaluation **11**

5 Conclusion **13**

1 Notations

In the following:

- $\mathbb{N}^0$ denotes the set of integers greater than or equal to 0,
- Double braces ($\{\!\{\cdots\}\!\}$) denote multisets,
- $|P|_a$ denotes the number of occurences of a in the multiset P,
- $[\![a, b]\!]$ denotes the set $\{a, a+1, \dots, b\}$.

2 Overview of FuncTion's Widening Operator

2.1 Introduction

Non-terminating programs have caused a number of problems in the past (see [3], [6]). Thus, it is important to guarantee that software has no termination issue, to avoid such bugs, that can sometimes even be used for denial-of-service attacks.

One way to prove properties of programs automatically is abstract interpretation [4], which allows to reason about the behaviour of programs in a formal and general manner. It is a *static analysis* method (other methods being used to check behaviour of programs being *deductive methods* and *model checking*). Static analysis methods, like abstract interpretation, do not need any user intervention, which comes at the cost of allowing *false positives* (correct programs being reported incorrect), but they still disallow *false negatives.* Moreover, there has been a lot of research advances on the topic of using abstract interpretation to prove termination of programs in recent years.

FuncTion is a tool that uses abstract interpretation to prove termination of C programs. As all abstract interpreters, FuncTion relies on the widening operator, an operator to accelerate convergence to compute post-fixpoints. We will briefly present what the widening operator in use in FuncTion is below.

2.2 The Decision Trees Abstract Domain

First, we present the decision trees abstract domain, which is the one used by FuncTion. It abstracts ranking functions, i.e. elements from $\mathcal{E} \rightharpoonup \mathbb{O}$, that map an environment to an ordinal greater than or equal to the number of program steps remaining.

The abstract values of this domain are binary trees, whose nodes are affine constraints on the variables, and whose leaves are of the form $\sum_{i=0}^{N} \omega^i \cdot f_i$, where N is a parameter of the analysis, and the f_i are affine functions with coefficients in $\mathbb{Z}$. Leaves can also be one of the special values $\bot_F$ and $\top_F$, which have special meanings. We will write F to denote the set of leaves, and $\widetilde{F} = F \setminus \{\bot_F, \top_F\}$ to denote the set of defined leaves.

The concretization function is easily computed: to compute $\gamma_T(t)(\rho)$, where t is an abstract value (and thus, a tree), and ρ an environment, follow the path decided by the constraints in the nodes, by going left or right depending on whether the constraint is satified in ρ, until getting to a leaf. If that leaf if either $\bot_F$ or $\top_F$, then $\gamma_T(t)(\rho)$ is undefined, otherwise, the value of $\gamma_T(t)(\rho)$ is the value of the leaf, evaluated in ρ (see Figure 1 for an example).

Note that the functions obtained have values in $\mathbb{O}$, the ordinals, and not only $\mathbb{N}$. The ordinals were introduced by Cantor in [2]. We will only use ordinals smaller than ω^ω in the following, and the property that any decreasing sequence of ordinals is finite.

Up to now, $\bot_F$ and $\top_F$ seem to have the same meaning. The difference in what they mean is as follows: $\bot_F$ represents a value that has not been computed yet, while $\top_F$ reprensents a value that has been computed, but for which termination is unknown.

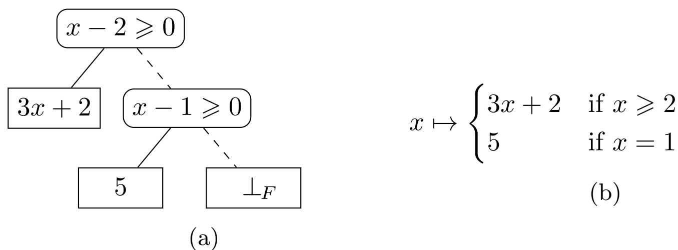


Figure 1: A decision tree T shown in (a), and the ranking function it represents $\gamma_T(t)$ in (b).

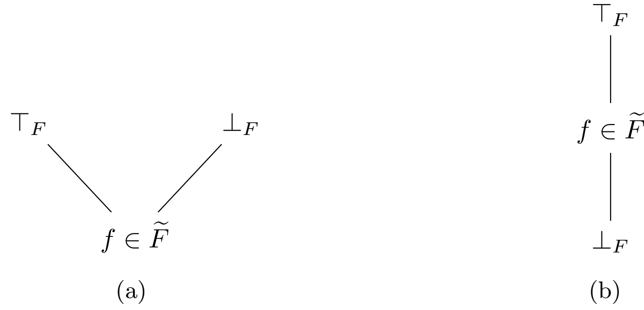
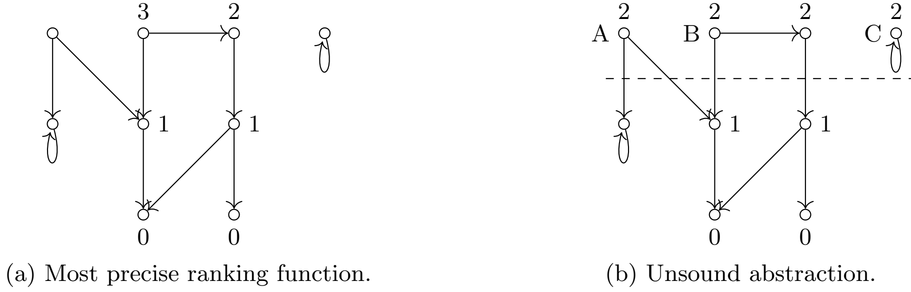


Figure 2: Hasse diagrams for the comparison of defined with undefined functions, for approximation order (a), and computational order (b).



(a) Most precise ranking function.

(b) Unsound abstraction.

Figure 3: The most precise ranking function (a) is unsoundly abstracted by (b). Case A: non-terminating loop outside over-approximated domain; case B: ranking function under-approximated; case C: non-termination loop inside over-approximated domain.

We thus have two different orders on leaves: the *computational order*, $\sqsubseteq_F$, for which $\bot_F$ is smaller than any function, and any function is smaller than $\top_F$, and the *approximation order*, $\preccurlyeq_F$, for which both $\bot_F$ and $\top_F$ are greater than any function, and are incomparable (see Figure 2). In both cases, f_1 is smaller than f_2 if it is smaller in all points of their domain, assuming those are equal. In the following, we will consider the domain of leaves to be implicit when doing operations of them.

Comparing trees is then straightforward, but relies on *tree unification*: given two trees, it is possible to add nodes to them so that the trees have the same nodes, and are equivalent to their previous versions. By keeping the constraints sorted in the trees, keeping the invariant that the smallest constraint is the root of the tree, the unification algorithm is straightforward. Once the trees are unified, comparing trees is immediate: one tree is smaller than another for according to an order (either computational of approximation) if and only if all its leaves are smaller according to this order than the corresponding leaf in the other tree, once the trees have been unified.

2.3 The Widening Operator on Decision Trees

We will now briefly describe the widening operator on decision trees; the curious reader that will want more detail or the definition of the other operators (that are less important for this report) can refer to [5], Ch. 5.

The widening operator tries to predict a value of the ranking function that will be stable: for that, we allow it more freedom than traditional widening operators, and we will allow it to temporarily over-approximate the domain of termination, or to under-approximate the value of the ranking function, but we will ensure that if it does that, it will not stop iterating while those problems have not been resolved.

Thus, the iterations we will use for widening will be the following:

$$y_0 \stackrel{\text{def}}{=} \bot_T$$
$$y_{n+1} \stackrel{\text{def}}{=} \begin{cases} y_n & \text{if } \phi(y_n) \sqsubseteq_T y_n \land \phi(y_n) \preccurlyeq_T y_n \\ y_n \triangledown_T \phi(y_n) & \text{otherwise} \end{cases}$$

Note how the usual stopping condition has been strengthened: this is to force the discrepancies caused by the widening operator to get resolved before iteration stops.

As it can be seen on figure 3, several cases can happen when the ranking function is incorrectly predicted: the predicted value of the ranking function can be too low for some states (e.g. case B), or some states from which the program does not terminate are included in the domain of the ranking function (e.g. case A, where the loop is outside the domain of definition of the unsound ranking function, or case C, where it is inside it).

Therefore, the widening operator will have to solve these discrepancies. Here is precisely how the widening operator works, to compute $t_1 \triangledown_T t_2$:

1. Check for case A: once the trees are unified, for each leaf f_2 in t_2, with corresponding leaf f_1 in t_1, check if $f_1 \sqsubseteq_F f_2$; if it is not the case, replace f_2 by $\top_F$.

   Indeed, in case A, the abstract ranking function will become $\bot_F$ in t_2 and therefore will fail the check. We then turn the leaf into $\top_F$ to avoid that point from being incorrectly reconsidered terminating again.

2. Perform *left unification* on t_1 and t_2: this step modifies t_2 to remove all its nodes that do not appear in t_1, unifying the two subtrees on its left and on its right.

   The objective of this step is to guarantee termination of the analysis, by bounding the number of possible segments.

3. Check for cases B and C: for each leaf f_2 in t_2, with corresponding leaf f_1 in t_1, if f_1 is defined (i.e neither $\top_F$ nor $\bot_F$) and $f_2 \not\preccurlyeq_F f_1$, then replace f_2 with $\top_F$. This is actually a widening on leaves, which result we will denote $f_1 \triangledown_F f_2$.

   If we are in cases B or C, the check will fail, thus replacing non-stable values by $\top_F$, which is stable (a bit like the widening on intervals, that sets non-stable bounds to $\pm\infty$ [4]).

4. Finally, extend each leaf in t_2 that was not defined in t_1 towards the adjacent segments in t_1.

   The rationale behind this step is to try to predict stable values for those segments, to avoid them being set later to $\top_F$ if they are not.

Note that, given the conditions we added to stop the iterations, we have the following property: *when* the widening iterations stop, then the result will be sound, whatever the definition of the widening operator is. We will use this property when improving the widening operator: this will ensure that it will still be correct.

3 Improving the Widening Operator

3.1 Retrying when Prediction was Incorrect

As we have seen previously, the widening operator would give up in case B, because the abstract ranking function was not stable, and the unstable leaf would be therefore replaced by $\top_F$. This causes an important loss of precision. Consider for instance Figure 4: here, the extend step correctly predicted the resulting value on two of the segments, but not on the third. When doing an iteration, it can be seen that the value on this segment is not stable, and is thus replaced by $\top_F$ by the original widening.

A simple and yet very powerful improvement is then to use the value in t_2 instead of $\top_F$ when the check in step 3 fails. However, doing this without caution causes the analysis to become non-terminating for a number of programs. Thus, we instead allow this operation to take place for the b first iterations of widening only, and then revert to the old behaviour. The results of that improvement on the preivous program can be seen in Figure 4d.

That change doesn't interface well with ordinals as-is: indeed, an unstable bound will grow, but with values in $\mathbb{N}^0$, and with then finally get turned into $\top_F$. However, there might be a stable bound, that is greater than ω, but which is not found.

We will therefore give the widening a certain number of *jokers*: the property we now want to enforce is that when the widening is given no jokers, it respects the traditional property of termination; this property is not necessary if the widening is given a positive number of jokers. Thus, the number of jokers can evolve arbitrarily during the successive widening iterations, but must be stationary to zero to ensure termination of the analysis. However, the widening operator will assume a lower number of jokers means it should do less precise operations, until zero jokers which mean it should use a widening that guarantees termination.

Let us write:

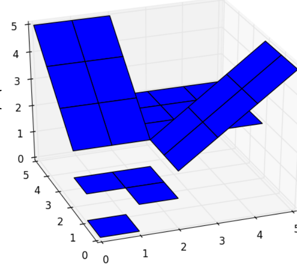
$$f_1 \triangledown_N f_2 = \begin{cases} f_2 & \text{if } f_2 \preccurlyeq f_1 \text{ or } f_1 \in \{\top, \bot\} \\ \top & \text{otherwise} \end{cases}$$

The current widening on leaves is then, if both leaves are defined:

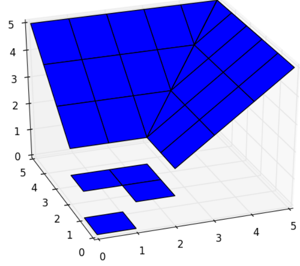
$$\left(\sum_{i=0}^{N} \omega^i \cdot f_1^i\right) \triangledown_F \left(\sum_{i=0}^{N} \omega^i \cdot f_2^i\right) = \begin{cases} \sum_{i=0}^{N} \omega^i \cdot \left(f_1^i \triangledown_N f_2^i\right) & \text{if } \forall i \in [\![1, N]\!], f_1^i \triangledown_N f_2^i \neq \top \\ \top_F & \text{otherwise} \end{cases}$$

```
int main() {
    int x, y;
    while (x > 0 || y > 0) {
        x--;
        y--;
    }
}
```

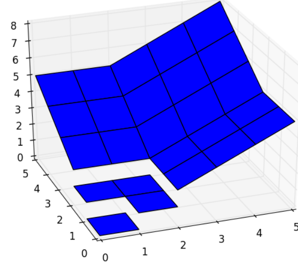
(a) A simple program.



(b) The ranking function inferred after the first widening iteration; not a correct ranking function.



(c) The best ranking function for that program.



(d) The ranking function inferred after a second step when using the retrying improvement.

Figure 4

The improvement, with jokers, is then defined by:

$$\mathbf{aux}\ i\ \langle f_1^1, f_1^2, f_1^3, \dots \rangle\ \langle f_2^1, f_2^2, f_2^3, \dots \rangle = \begin{cases} \langle 0,\ \mathbf{aux}\ (i-1)\, \langle f_1^2, f_1^3, \dots \rangle\, \langle f_2^2 + 1, f_2^3, \dots \rangle \rangle & \text{if } z = \top \\ \langle z,\ \mathbf{aux}\ (i-1)\, \langle f_1^2, f_1^3, \dots \rangle\, \langle f_2^2, f_2^3, \dots \rangle \rangle & \text{otherwise} \end{cases}$$

$$\text{where } z = \begin{cases} f_1^1 \nabla_N f_2^1 & \text{if } i > 0 \\ f_2^1 & \text{otherwise} \end{cases}$$

$$\left(\sum_{i=0}^{N} \omega^i \cdot f_1^i \right) \overset{k}{\nabla}_F \left(\sum_{i=0}^{N} \omega^i \cdot f_2^i \right) = \begin{cases} \sum_{i=0}^{N} \omega^i \cdot f_3^i & \text{if } \forall i > N, f_3^i = 0 \\ \top_F & \text{otherwise} \end{cases}$$

$$\text{where } \langle f_3^1, f_3^2, f_3^3, \dots \rangle = \mathbf{aux}\ (N+1-k) \left\langle f_1^1, f_1^2, \dots, f_1^N, 0, 0, \dots \right\rangle \left\langle f_2^1, f_2^2, \dots, f_2^N, 0, 0, \dots \right\rangle$$

What $\overset{k}{\nabla}_F$ thus does is to use the widening operator for the low-order terms, then switch to just keeping the second term for the high-order ones, replacing any $\top$ that might appear by increasing the following ordinal. It is a bit like long addition of numbers, with $\top$ appearing meaning we should propagate a carry.

Lemma 1. *$\overset{k}{\nabla}_F$ is well defined and computable.*

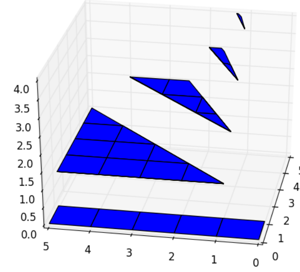
Proof. We remark that $\forall i \in \mathbb{Z}, \mathbf{aux}\, i\, \langle 0, 0, \dots \rangle\, \langle 0, 0, \dots \rangle = \langle 0, 0, \dots \rangle$. We then have, $\mathbf{aux}\, i\, \langle 0, 0, \dots \rangle\, \langle 1, 0, 0, \dots \rangle$ well-defined and computable as well, since it is equal to $\langle 1, 0, 0 \dots \rangle$ if $i < 0$, and i is otherwise decreasing.

This implies that, after $N+1$ recursive calls of **aux**, we are in one of the two previous cases, and the result claimed follows. □

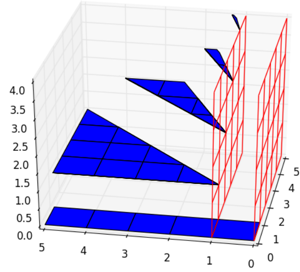
We can then describe the new widening on trees as a widening with jokers as well: to compute $t_1 \overset{k}{\nabla}_T t_2$, do the same thing as before, but replacing

```
int main() {
    int x, y;
    if (y > 0) {
        while (x > 0) {
            x -= y;
        }
    }
}
```

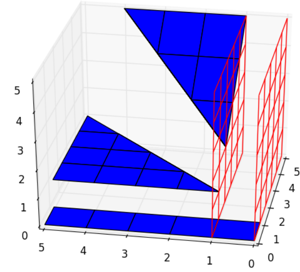
(a) Another simple program.



(b) The ranking function inferred just before widening.



(c) The constraints added by the evolving rays.



(d) The ranking function inferred with the help of evolving rays.

Figure 5

$$\begin{cases} \nabla_F \text{ by } \overset{\lceil \frac{k}{b} \rceil}{\nabla}_F & \text{if } b > 0 \\ \nabla_F \text{ by } \overset{0}{\nabla}_F & \text{otherwise} \end{cases}$$

Here, b is another parameter of the analysis, that controls how many iterations we try again at each ordinal level. The iteration sequence is then replaced by the following one:

$$y_0 = \bot_T$$
$$y_{n+1} = \begin{cases} y_n & \text{if } \phi(y_n) \sqsubseteq_T y_n \wedge \phi(y_n) \preccurlyeq_T y_n \\ y_n \overset{r-n}{\nabla}_T \phi(y_n) & \text{otherwise} \end{cases}$$

where r is the initial number of jokers, $b \cdot (N + 1)$

3.2 Evolving Rays

Another problem with the widening operator is that is never infers new constraints: thus, a program as the one in Figure 5 is not proven terminating, because the test that would be needed ($y \geqslant 1$) is never added to the tree. Unfortunately, this kind of problem happens in a lot of programs.

A solution is then to change the left-unification phase (step 2 of the definition of the widening) to add these new constraints. Two problems arise there:

- How to decide which constraints to add to the tree?
- How to make sure the widening iterations will still terminate?

We will use a method inspired by the evolving rays presented in [1] for the former. We will recall the

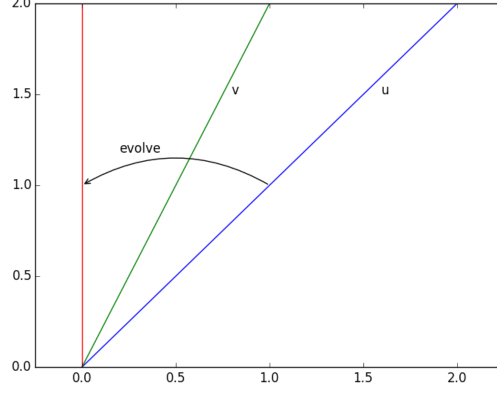


Figure 6

definition of **evolve** that is presented in it:

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

The effect of **evolve** can be seen on Figure 6.

We also remember that any constraint can be put into a canonical form, where the greatest common divisor of all its coefficents (except the constant one) is 1. All the constraints in the tree are in canonical form, but **evolve** can create constraints that are not. Thus, we need to simplify then, and we will note **canonical** the operation that puts a constraint into its canonical form. We remind its definition, which is simply:

$$\textbf{canonical}\left(\sum_i a_i x_i \geqslant c\right) = \sum_i \frac{a_i}{g} x_i \geqslant \left\lceil \frac{c}{g} \right\rceil$$
$$\text{where } g = \gcd_i a_i$$

The idea then is to add constraints to the tree, but not too many of them: we first compute a set of allowed constraints. For that, we define **constraints**(t) the multiset of all constraints in t and their negation, and **adjacent**(t) the multiset of pairs of constraints or their negation in t that are contained in nodes where one is an ancestor of the other. Formally:

$$\textbf{constraints}(\textsc{Leaf}(f)) = \{\!\{\}\!\}$$
$$\textbf{constraints}(\textsc{Node}(c, l, r)) = \textbf{constraints}(l) \cup \textbf{constraints}(r) \cup \{\!\{c, \neg c\}\!\}$$

$$\textbf{adjacent}(\textsc{Leaf}(f)) = \{\!\{\}\!\}$$
$$\textbf{adjacent}(\textsc{Node}(c, l, r)) = \textbf{adjacent}(l) \cup \textbf{adjacent}(r) \cup \left\{\!\!\left\{ (c_1, c_2) \;\middle|\; \begin{array}{c} c' \in \textbf{constraints}(l) \cup \textbf{constraints}(r), \\ (c_1, c_2) \in \{\!\{(c, c'), (c', c), (\neg c, c'), (c', \neg c)\}\!\} \end{array} \right\}\!\!\right\}$$

and we then set:

$$P = \left\{\!\!\left\{ \textbf{canonical}\left(\sum_i c_i x_i \geqslant e\right) \;\middle|\; \begin{array}{c} \left(\sum_i a_i x_i \geqslant \alpha,\ \sum_i b_i x_i \geqslant \beta\right) \in \textbf{adjacent}(t), \\ c = \textbf{evolve}(a, b),\ e \in \{0, 1\} \end{array} \right\}\!\!\right\}$$
$$\textbf{allowed}(t) = \textbf{constraints}(t) \cup \left\{ c \in P \;\middle|\; |P|_c \geqslant p \right\}$$

Here, p is another parameter of the analysis, the *evolve threshold*; setting p to be equal to the number of iterations before widening is usually a good choice. It controls whether new constraints should be added or

not: only those that appear more than p times are added, which the others are ignored. The only constraints we will keep in the resulting tree at the end of step 2 will be the elements of $\mathbf{allowed}(t_1)$. All others, i.e., the nodes containing elements of $X = \mathbf{constraints}(t_2) \setminus \mathbf{allowed}(t_1)$, will be removed and replaced by other nodes containing constraints that are allowed. We thus define:

$$P = \left\{\!\!\left\{ \left(\sum_i a_i x_i \geqslant \alpha,\ \mathbf{canonical}\left(\sum_i c_i x_i \geqslant e \right) \right) \middle| \begin{array}{c} \left(\sum_i a_i x_i \geqslant \alpha,\ \sum_i b_i x_i \geqslant \beta \right) \in \mathbf{adjacent}(t_2), \\ c = \mathbf{evolve}(a,b),\ e \in \{0,1\} \end{array} \right\}\!\!\right\}$$

$$\mathbf{replace}(t_1, t_2) = c_2 \in X \mapsto \left\{ c \in \mathbf{allowed}(t_1) \middle| |P|_{(c_2,c)} \geqslant p \right\}$$

The function $\mathbf{replace}(t_1, t_2)$ therefore maps each of the constraints that is to be removed to a set of constraints that will replace it. What the left unification algorithm then does instead of deleting constraints as previously is to replace all of them by their image via $\mathbf{replace}(t_1, t_2)$; removing then redundant constraints, and joining the leaves when needed.

To guarantee termination, we define the *completion* of a set of constraints S to be:

$$\mathbf{complete}(S) = S \cup \left\{ \mathbf{canonical}\left(\sum_i c_i x_i \geqslant e \right) \middle| \begin{array}{c} \left(\sum_i a_i x_i \geqslant \alpha \right) \in S, \left(\sum_i b_i x_i \geqslant \beta \right) \in S, \\ c = \mathbf{evolve}(a,b), e \in \{0,1\} \end{array} \right\}$$

We then define:

$$\mathbf{complete^*}(S) = \bigcup_{n \in \mathbb{N}^0} \mathbf{complete}^n(S)$$

It can be seen that $\mathbf{complete^*}(S)$ is finite if S is finite. Indeed, we have:

$$\mathbf{complete^*}(S) \subseteq S \cup \left\{ \mathbf{canonical}\left(\sum_i b_i x_i \geqslant e \right) \middle| \left(\sum_i a_i x_i \geqslant \alpha \right) \in S, b_i \in \{0, a_i\}, e \in \{0,1\} \right\}$$

Lemma 2. *If t_3 is the result of the new algorithm of left unification, then:*

$$\mathbf{constraints}(t_1) \subseteq \mathbf{constraints}(t_3) \subseteq \mathbf{complete}(\mathbf{constraints}(t_1))$$

Proof. We have $\mathbf{constraints}(t_1) \subseteq \mathbf{constraints}(t_2)$, because we perform tree unification in step 1 of widening, which modifies t_2 and preserves t_1, therefore adding to t_2 any constraint that might be in t_1 and that were not in it yet. As $\mathbf{constraints}(t_1) \subseteq \mathbf{allowed}(t_1)$, all these constraints will be kept.

Moreover, we have $\mathbf{allowed}(t_1) \subseteq \mathbf{complete}(\mathbf{constraints}(t_1))$, by definition of **evolve**, **allowed** and **complete**, thus implying $\mathbf{constraints}(t_3) \subseteq \mathbf{complete}(\mathbf{constraints}(t_1))$. □

We now have $\mathbf{constraints}(t_1), \mathbf{constraints}(t_1 \triangledown_T t_2), \ldots$ which is an increasing chain in a finite space, $\mathbf{complete^*}(\mathbf{constaints}(t_1))$, and is therefore stationary. Once it becomes constant, the left-unification step is exactly the same as the previous one, therefore causing termination of the analysis.

3.3 Extending Towards Relevant Segments

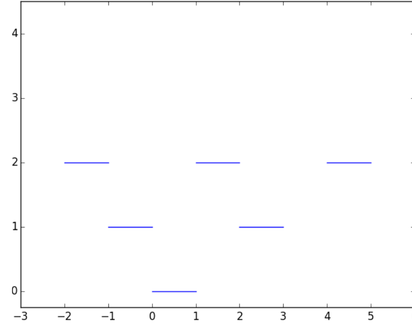
Yet another problem is the extend step (step 4). Indeed, extending towards adjacent segments is a good heuristic for many programs, but suffers from several problems:

- Sometimes, an environment in a segment at one iteration becomes an environment in a non-adjacent segment after another iteration. Figure 7 is such an example.
- Dividing a segment into multiple ones, as the evolving rays do, causes some of the new segments to lose the benefit from the extend step.

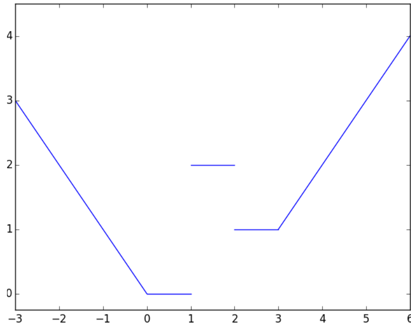
The idea to solve these problems is therefore to find which segments to extend towards instead of extending only towards adjacent segments. Also, note that this step is only here to improve precision of the analysis: completely skipping it would not affect the soundness of it, but the termination of very few programs will be proven. Actually, we may even choose the segments to extend towards in any way we like: only the precision of the analysis will be affected, not its termination or its soundness.

```
int main() {
    int x;
    while (x != 0) {
        if (x > 0) {
            x -= 2;
        } else {
            x += 1;
        }
    }
}
```

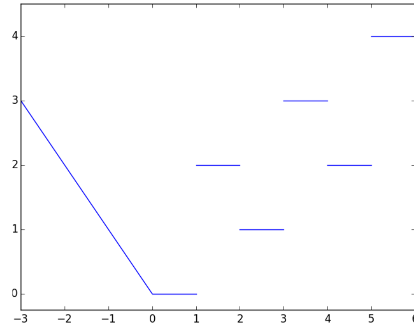
(a) Yet another simple program.



(b) The ranking function inferred just before widening.



(c) The incorrect ranking function inferred after the first widening iteration.



(d) The most precise ranking function for that program.

Figure 7

Thus, we would like to extend each segment only towards the segments that contained one of their environments on the previous iteration. For that, for each segment, we simply look at their abstract domain. Next, we perform a step of *forward* propagation of this segment on the loop body, to get a new abstract domain, D. We then simply extend our segments towards the segments that intersect D, instead of towards the adjacent ones. Note that this only works if we indeed have a loop body to do this on: other uses of the widening operator will need to keep the previous version.

While helping for some programs, this also has a default: when the variables that are concerned are modified a lot in the loop (for example, if x becomes $-x$), this can cause extend steps that will actually make the analysis *less* precise than it was before. Thus, proving termination should be tried both without and with this option.

Note that, the result does not comply to the definition of a widening operator anymore; in fact, we can get different results even if both arguments are equal if the loop body is different! Thus, to formalize that, this widening takes yet another argument, a function from polyhedra to polyhedra that does forward propagation of its argument on the loop body.

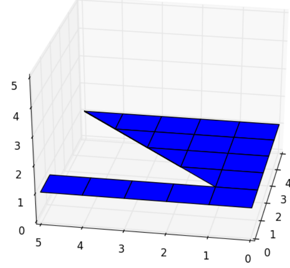
3.4 Putting it all together

We now formalize these improvements together. Our widening is a *widening with jokers and extra information*, that is an operator ∇_T that given $k \in \mathbb{N}^0$, $x, y \in T$ such that $x \sqsubseteq_T y$, and $a \in A$, where A is the set where extra information lives in, returns a value in T satisfying:

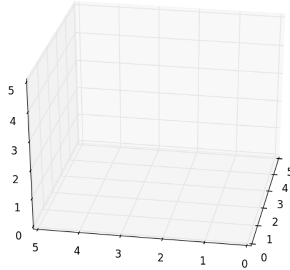
- $x \overset{k}{\nabla}_T(a)\; y \sqsubseteq_T y$,

```
int main() {
    int x, oldx;
    while (x > 0 && x < oldx) {
        oldx = x;
        x = ?;
    }
}
```

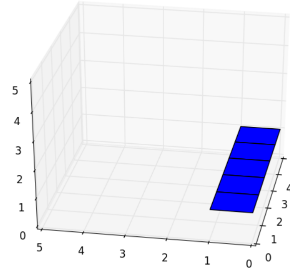
(a) A program with non-determinism.



(b) The ranking function inferred just after the `x = ?` assignment.



(c) The ranking function inferred before the assignment with the original backwards assignment operator. This is $\bot_T$.



(d) The ranking function inferred before the assignment with the improved backward assignment operator.

Figure 8

- The sequence defined by:

$$x_0 = \bot_T$$
$$x_{n+1} = \begin{cases} x_n & \text{if } y_n \sqsubseteq_T x_n \wedge y_n \preccurlyeq_T x_n \\ x_n \overset{k_n}{\nabla}_T(a_n)\ y_n & \text{otherwise} \end{cases}$$

  is stationary for all choices of $a_n \in A$, $k_n \in \mathbb{N}^0$ such that k_n is ultimately zero, and $y_n \in T$ such that $x_n \sqsubseteq_T y_n$.

In our case, A is the set of functions from abstract domains to abstract domains, and we set $y_n = x_n \sqcup_T \phi(x_n)$, a_n is always equal to the function that maps a polyhedron to its post-image by the loop body, and $k_n = \max(0, b \cdot (N+1) - n)$.

3.5 Other Improvements

I also made various improvements to other parts of FuncTion that were not the widening operator, but that made it able to prove termination of a few more test cases:

- Allow rational coefficients to be used in ranking functions. The ranking functions now have to decrease by at least 1 each step instead of being simply decreasing, to ensure that the existence of a ranking function implies termination. Indeed, any decreasing sequence of ordinals is finite, but this is not the case for rationals. However, if we require that all ranking functions decrease by at least 1 each step, then the integer parts of these functions make a decreasing sequence of ordinals, showing these are finite as well.
- Fix the ordinal comparison function, which was previously telling that $\omega + 1 \leqslant \omega$. This allowed me to re-enable the use of ordinals in the widening operator, which was disabled as a workaround to this bug.

	Success	Timeout	Time	Time (w/o timeouts)
Original version (i.e. $b = 0$, evolving rays disabled)	104	4	7.9s	3.0s
$b = 0$	101	10	19.1s	7.0s
Evolving rays disabled	138	6	10.6s	3.2s
New version	136	15	28.9s	10.9s
With refining	139	14	26.9s	10.1s
Extending towards the relevant segments	134	12	27.4s	13.1s
Extending towards the relevant segments and refining	123	11	26.0s	12.9s
Improved backwards assignment operator	133	49	74.2s	16.8s
Improved bwd. assignment operator and refining	132	54	81.7s	18.9s
Rational coefficients	133	12	30.8s	16.8s
Rational coefficients and refining	135	12	28.6s	14.4s
Widening delay of 6 iterations	125	37	66.4s	24.1s
Widening delay of 6 it. and improved bwd. assignment	117	92	126.8s	20.4s
Automated version	195	6	33.7s	26.9s

Figure 9: Results of comparing FuncTion with various options. Unless otherwise specified, all tests are with $b = 10$, $N = 2$, evolving rays enabled and $p = 3$, the polyhedra domain and widening delay of 3 iterations.

- Add an option that improves the precision of the backwards assignment operator, at the cost of trees becoming a lot bigger. Instead of backwards propagating each constraint separately, it propagates each segment backwards, and then joins the resulting segments. This was useful in particular for programs such as the one in figure 8.

4 Experimental Evaluation

The improved FuncTion implementation was evaluated on 242 terminating C programs from the *4th International Competition on Software Verification (SV-COMP 2015)*, removing test cases that made use of recursion, arrays or pointers. The experiments were performed on a system with a 3.20GHz 64-bit dual-core CPU (Intel i5-3470) and 8GB of RAM, running Debian 3.16.7-ckt25-2 (2016-04-08).

Figure 9 shows the results obtained with different options. All results were obtained with the polyhedra domain, ordinal limit $N = 2$, widening delay of 3 iterations, evolve threshold $p = 3$ and $b = 10$ (so, 30 jokers initially), unless otherwise specified. The timeout used was 300 seconds, after which test cases were interrupted.

The automated version executes FuncTion with 12 various option combinations successively, with a time limit of 25 seconds each. If any of those analysis proves termination, then the program is reported as terminating and the following analysis are not run. The common options are, as before, the polyhedra domain, $b = 10$, $p = 3$ and $N = 2$. The specific options used are, in the order they are tried: evolving rays disabled; the default version; refining; extending towards the relevant segments; rational coefficients; ordinal limit $N = 3$ and widening delay of only 2 iterations; evolving rays disabled and extending towards relevant segments; refining and extending towards relevant segments; evolving rays disabled and widening delay of 6 iterations; evolving rays disabled, widening delay of 6 iterations and improved backwards assignment operator; improved backwards assignment operator; and finally refining, improved backwards assignment operator and widening delay of only 2 iterations. The rationale behind this order is that disabling evolving rays makes the analysis quite fast, even if it is not able to prove termination, and still proves a lot of programs. Then, the evolving rays are enabled, and this proves more programs. The improved backwards assignment operator useful mainly for programs with non-determinism that otherwise do not even get a widening operation (because a fixpoint is found before). Since the analysis of those programs is very fast otherwise, and the analysis of other programs is usually slowed a lot by this backwards assignment operator, it is only used after all other possibilities have been tried.

The refine option is one we have not yet talked about: it does a step of forward analysis to find reachable states before using this information during the backwards analysis to reduce the domain of the trees, to improve precision a bit.

As it can be seen, retrying when prediction was incorrect is the most important improvement: with it alone, the termination of more than 30 more programs is proven. The evolving rays improvement does not

	Successes
Evolving rays disabled (■)	138
Default options	17
With refining	11
Extending towards the relevant segments (▲)	6
Rational coefficients	4
Widen. delay 2, $N = 3$	1
■ + ▲	0
Refining + ▲	1
Widen. delay 6 + ■	5
Widen. delay 6 + ■ + ⧫	9
Improved backwards assignment operator (⧫)	0
Refining + widen. delay 2 + ⧫	3

Figure 10: Number of successes proven by each option combination first in the automated mode, in the order they are tried.

	Original version				New version			
	□	△	+	−	□	△	+	−
Original version	0	0	104	138	9	41	95	97
$b = 0$	10	13	91	128	0	35	101	106
Evolving rays disabled	34	0	104	104	19	17	119	87
New version	41	9	95	97	0	0	136	106
With refining	45	10	94	93	13	10	126	93
Extending towards the relevant segments	42	12	92	96	9	11	125	97
Extending towards the relevant segments and refining	38	19	85	100	14	27	109	92
Improved backwards assignment operator	42	13	91	96	11	14	122	95
Improved bwd. assignment operator and refining	50	22	82	88	19	23	113	87
Rational coefficients	44	15	89	94	4	7	129	102
Rational coefficients and refining	45	14	90	93	14	15	121	92
Widening delay of 6 iterations	32	11	93	106	7	18	118	99
Widening delay of 6 it. and improved bwd. assignment	32	19	85	106	11	30	106	95
Automated version	91	0	104	47	59	0	136	47

Figure 11: Comparing FuncTion with various option combinations. □ is the number of test cases proven only by the combination on the left, △ only by the combination on the top, + by both and − by none.

seem very helpful when looking at these results; however, what happens is that it proves termination of a lot more programs, but fails to prove termination of a lot of other programs which the version without evolving rays could prove. All in all, this makes 17 additional programs whose termination can only be proven with the help of evolving rays. It can also be seen that evolving rays does make the analysis slower: we attribute this slowdown to the fact that evolving rays cause the trees to become a lot bigger. Even worse is the slowdown caused by the improved backwards assignment operator: indeed, it also causes the trees to become a lot bigger, but it is used a lot more than the widening operator. Rational coefficients cause a slight slowdown as well, which is however less important than the one caused by the other changes. A higher widening delay slows the analysis a lot – more than usual with abstract interpretation, because the trees become very large before the widening starts, and thus has to work with these very large trees.

While all the changes made look like they are useless, except for the retrying when prediction was incorrect one, the automated version shows this in not true: the improvements it makes over the other versions are because the different option combinations prove different programs. Figure 11 shows the comparison of the different options against the previous version and the current one, and Figure 10 shows the contribution of the various option combinations towards the total number of successes in the automated version. Finally, Figure 12 shows more precisely which test cases are solved by which options.

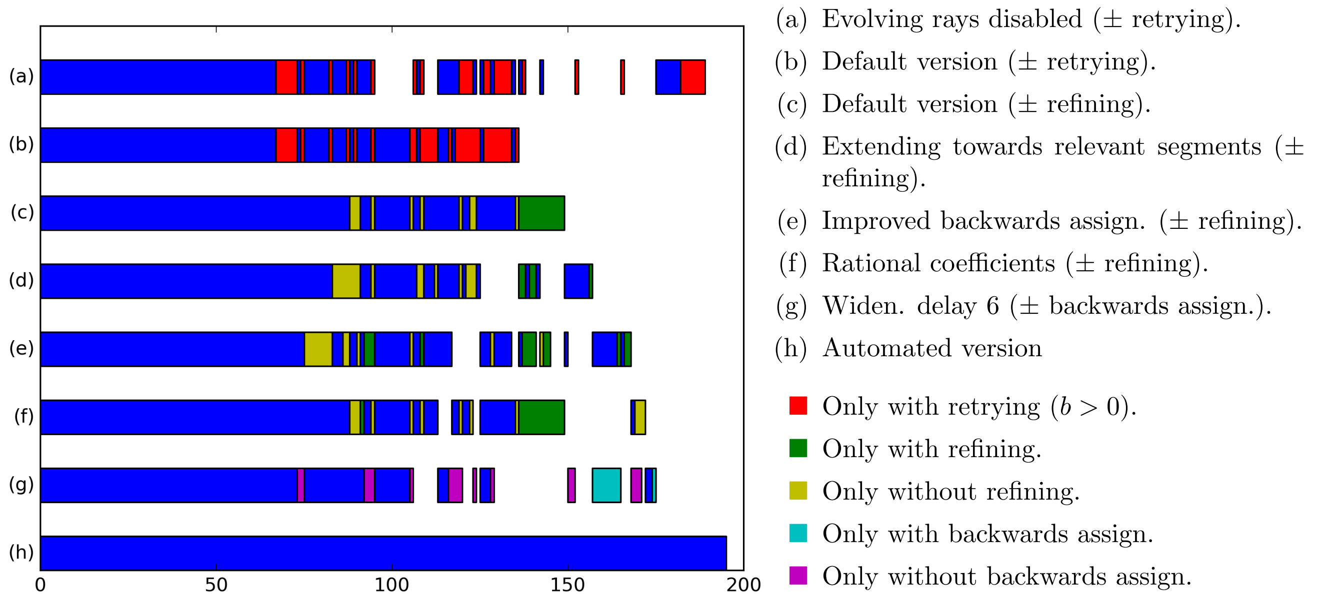


Figure 12: Overview of the tests solved for each option combination. Each coloured bar corresponds to a successful test, points on the same vertical line correspond to the same test.

5 Conclusion

The experimental evaluation shows that although the various changes improve precision, precision is also lost as the widening operator is non-monotonic. Moreover, this additional precision costs a lot of time. Thus, these options that improve precision need to be able to individually be enabled or disabled on a case-by-case basis. However, we have seen that trying to execute FuncTion both with and without these improvements, with a time limit, significantly improves the performance, as positive results are almost always obtained fast, and after some time, only negative results are reported.

A possible improvement would be to try to detect which segments to keep and which to merge, to keep the size of the trees more reasonable. Another would be to improve the operator currently used for extending, to make it predict better ranking functions, and maybe to choose the segments towards which to extend better.

References

[1] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1):28–56, 2005.

[2] G. Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 49(2):207–246, 1897.

[3] D. Coldewey. Zune bug explained in detail. https://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/, 2008. [Online; accessed 18-July-2016].

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[5] C. Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs*. PhD thesis, École Normale Supérieure, 2015.

[6] J. Zander. Update on Azure Storage Service Interruption. https://azure.microsoft.com/en-us/blog/update-on-azure-storage-service-interruption/, 2014. [Online; accessed 18-July-2016].