Robust, Iterative Dynamically Deployed Static Analysis

Sebastian Gfeller

May 15, 2009

1 Introduction

Many interesting properties of programs can be found out statically before program execution. However, the future execution of most programs depend heavily on supplied configuration like command line arguments, preference files and initial user input.

We want to use this dynamically obtained information to perform a more precise static analysis over the running program.

Let's look at the following simple Java application:

```
class Example {
  static class A {
    int f;
    int g;
  }

  public static void main(String[] args) {
    Random r = new Random();
    A x = new A();
    x.f = Integer.valueOf(args[0]);
    x.g = r.nextInt();
    A y = new A();

    TM(x,y);

    while (x.f != 1) {
      if (x.f % 2 == 0) x.f = x.f / 2;
      else x.f := x.f * 3 + 1;
    }

    assert x.g == y.g;
  }
}
```

Where `TM(x,y)` is a complex method. As you can see, this program depends on various forms of input. One important question that we would like to answer is "Will the assertions in the future program execution hold". Without running the example program for some time, we will not know this. But already after having executed `TM(x,y)`, we have a very clear idea.

In the rest of this document, we

- Specify how we can use dynamically obtained state to drive static analysis using constant propagation for a flat heap as an example.

- Show how we can use a simple backward analysis to obtain error states
- Combine the forward and backward information to discharge program assertions earlier on

2 The language

A dynamic analysis requires that we can obtain program state and combine it with information about the currently executing code. In a fully compiled executable, we have lost the ability to reliably analyze execution. If we use the source code of a high-level language, mapping the state to the code becomes much more difficult, while analyzing execution becomes easy.

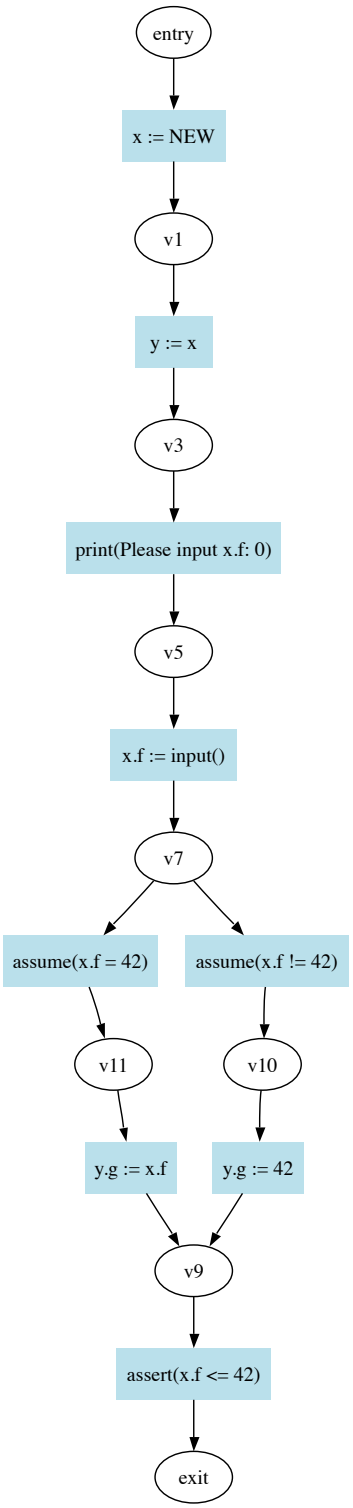
One intermediary solution is to look at programs written in a low-level bytecode that are then executed by a virtual machine. These can be easily abstracted by a control-flow graph.

We work on a CFG, the following program statements can occur:

$$
\begin{array}{rcl}
\text{statement} & ::= & x.f := \text{expression} \mid x := y \mid x := new \mid x.f := input() \mid \\
 & & print(\text{ string, expression }) \mid assume(\text{ formula }) \mid assert(\text{ formula }) \\
\text{formula} & ::= & \text{expression} = \text{expression} \mid \text{expression} \neq \text{expression} \mid \\
 & & \text{expression} \leq \text{expression} \mid \text{expression} \geq \text{expression} \mid \\
 & & \text{expression} < \text{expression} \mid \text{expression} > \text{expression} \\
\text{expression} & ::= & \text{atomic} \mid \text{expression} + \text{expression} \mid \text{expression} - \text{expression} \mid \\
 & & \text{expression} * \text{expression} \mid \text{expression}/\text{expression} \\
\text{atomic} & ::= & x.f \mid \mathbf{c}
\end{array}
$$

x, y variables, f, g fields.

2.1 Example


entry
x := NEW
v1
y := x
v3
print(Please input x.f: 0)
v5
x.f := input()
v7
assume(x.f = 42)
assume(x.f != 42)
v11
v10
y.g := x.f
y.g := 42
v9
assert(x.f <= 42)
exit

2.2 Representing a real-life language

The language presented here might not seem very realistic, and this is primarily because we are concentrating on a flat heap. Note, however, that we can make a sound conversion from a language with a richer heap structure and primitive values as follows:

Primitive values For all the local variables of primitive type, we can just represent them in one special object, sf. $x := 5$ becomes $sf.x := 5$. No magic here.

Object fields Now let us look at the case $x := y.f$. Our abstraction using relations does not let us model such cases. But when it comes to forward as well as backward analysis, it is sound to say

$$x := new$$

here.

Non-primitive operations on integers could be for example accessing a field-of-a-field, or calling a native function. In this case we cannot do more than to

$$x.f := *$$

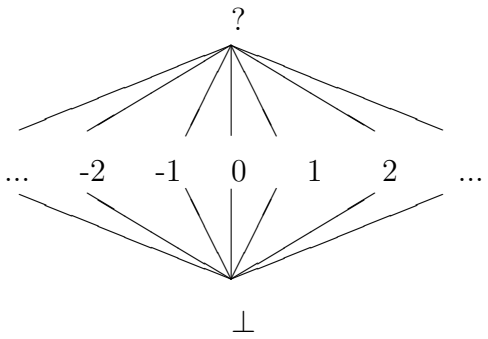
But note that if this is not sufficient, we cane retake analysis after this value was calculated.

3 Forward Analysis

Each variable maps to a store. Each store maps a field to a constant.

3.1 Standard Constant Propagation lattice

For constants, we use the following lattice:



where

$$x \sqcup y = \begin{cases} x & \text{if } x = y \text{ or } y = \bot \\ y & \text{if} x = \bot \\ ? & \text{otherwise} \end{cases}$$

$$x \sqcap y = \begin{cases} x & \text{if } x = y \text{ or } y = \top \\ y & \text{if } x = \top \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \bot & \sqsubseteq & x \\ x & \sqsubseteq & \top \\ x & \sqsubseteq & y \Leftarrow x = y \end{array}$$

3.1.1 Interpretation

For the entry point and initial store S_i, we have:

$$[\![entry]\!]_{\mathcal{S}} = \lambda f.c_f$$

Where c_f is the value of field f in the current state. For an S_i not an initial store, we have

$$[\![entry]\!]_{\mathcal{S}} = \lambda f.\bot$$

For a variable assignment, with S_{prev} being the value of store S in the previous state:

$$[\![x := y]\!]_{\mathcal{S}} = S_{prev}$$

For a new instance creation, if S is the store associated with instances at this label:

$$[\![x := \text{new}]\!]_{\mathcal{S}} = \lambda f.\bot$$

For every other store, nohing changes.
For a field assignment where xRS, x is related to S:

$$[\![x.f := a]\!]_{\mathcal{S}} = S_{prev}[f \rightarrow [\![a]\!]_{\mathcal{A}}]$$

For an unrelated field assignment:

$$[\![x.f := a]\!]_{\mathcal{S}} = S_{prev}$$

For an assume statement:

$$[\![\text{assume } cond]\!]_{\mathcal{S}} = \begin{array}{cc} S_{prev} & \text{if } [\![cond]\!]_{\mathcal{B}} \in \{1, \frac{1}{2}\} \\ \bot & \text{otherwise} \end{array}$$

For an arithmetic expression:

$$[\![a_1 * a_2]\!]_{\mathcal{A}} = \begin{cases} c_1 * c_2 & \text{if } [\![a_1]\!]_{\mathcal{A}} U_{prev} = c_1 \text{ and } [\![a_2]\!]_{\mathcal{A}} U_{prev} = c_2 \\ ? & \text{otherwise} \end{cases}$$

For $* \in \{+, -, \cdot, /\}$

$$[\![x.f]\!]_{\mathcal{A}} = \bigsqcup_{(x,S) \in R_{prev}} S(f)$$

And finally, for relations:

$$[\![a_1 \mathcal{R} a_2]\!]_{\mathcal{B}} = \begin{cases} \frac{1}{2} & \text{if } [\![a_1]\!]_{\mathcal{A}} =? \text{ or } [\![a_2]\!]_{\mathcal{A}} =? \\ [\![a_1]\!]_{\mathcal{A}} \mathcal{R} [\![a_2]\!]_{\mathcal{A}} & \text{otherwise} \end{cases}$$

For $\mathcal{R} \in \{=, \neq, \leq, <, >, \geq\}$

3.2 Store lattice

A store S is a function from a field name to a constant. $S_i : Fields \rightarrow Constants$. Stores are identified by their definition point i. We define $\sqcup$ as follows:

$$S_1 \sqcup S_2 = \lambda f.S_1(f) \sqcup S_2(f)$$

Also, define $S[f \rightarrow \mathbf{c}]$ to be the store where field f points to value $\mathbf{c}$.

$$S_1 \sqsubseteq S_2 \Leftrightarrow \forall f.\ S_1(f) \sqsubseteq S_2(f)$$

We will have multiple stores, one per program point and one per store (heap object) at the entry vertex, mapping to the current values.

$$\bot = \lambda f.\bot$$

$$\top = \lambda f.?$$

3.3 Relation between variables and stores

$$R \subseteq X \times S^{|H|+|L|}$$

Where X are the variable names, H are the stores at the entry point and L are the edges where an instance can be created.

$$\bot = \emptyset$$

$$\top = X \times S^{|H|+|L|}$$

Where $\sqsubseteq$ is $\subseteq$, $\sqcup$ is $\cup$ and $\sqcap$ is $\cap$.

3.3.1 Interpretation

For a variable assignment:

$$[\![x := y]\!]_{\mathcal{R}} = (R_{prev} \cup \{(x, S) | (y, S) \in R_{prev}\}) \setminus \{(x, S) | (y, S) \notin R_{prev}\}$$

For a new instance at label l:

$$[\![x := \text{new}]\!]_{\mathcal{R}} = (R_{prev} \cup \{(x, S_l)\}) \setminus \{(x, S) | S \neq S_l\}$$

Nothing changes for field assignment or assume.

3.4 Pointwise

Now we are able to define a lattice for a program point. It is just a product lattice between a relation R and $S^{|I|+|L|}$ stores:

$$((R, S^{|I|+|L|}), \sqsubseteq)$$

where

$$(R_1, S_{11}, \ldots, S_{1n}) \sqsubseteq (R_2, S_{21}, \ldots, S_{2n}) = R_1 \sqsubseteq R_2 \wedge \bigwedge_{i \in 1,\ldots,n} S_{1i} \sqsubseteq S_{2i}$$

$\sqcup$, $\sqcap$ are also defined pointwise.

3.5 Discharging Assertions

Now that we have propagated all constants, we can define how to discharge assertions. From the current program point on, we collect all assertions that will still be visited in the future. Each of those is then evaluated with the information obtained by the constant propagation. If this information is enough, the assertion is discharged.

4 A Rough Backward Analysis

The goal of the static analyses that are done here is to discharge some assertions earlier on. The constant propagation forward analysis works fine if assertions do not depend on values calculated further down in the program any more. However, it has some disadvantages:

- At each program point where analysis is desired, it has to be calculated again.
- It only takes into account equalities: Information about variable range is ignored.

If we use backwards analysis, we can take into account both points. Because backward analysis does not depend on the current state (modulo some factors like call graph information), we can do it incrementally. And using a reasonable abstract domain, we can take into account inequalities.

4.1 "Pushing up" assertions

To discharge assertions earlier on, we have to find conditions that, if triggered, may fire the assertion later on. The most exact condition that says that we won't fire any assertion is the weakest precondition at this program point. However, as the program is not loop-free, we might never find such a precondition. While there exist approaches using templates to guess invariants for such loops, instantiating these templates is quite slow for complex programs.

Also, a template-based approach is not fully automated - a property that is desirable for our analysis.

4.2 Finding a good domain

To find a good abstract domain for backward analysis, we have to take into account the form of assertions that we would like to handle. Some important arithmetic constraints are a non-zero check `assert(x.f != 0)` (to avoid division by 0) and non-negativity.

A seemingly suitable domain is the domain of polyhedra[2]. In this domain, constraints on the fields of the heap objects are defined as a conjunction of linear inequalities

$$a_1 x.f + a_2 y.f + a_3 x.g + a_4 y.g + \ldots \leq c$$

This is one of the first numerical domains proposed for abstract interpretation, but it still holds up to the task for which we need it. We assume that we can use a framework to deal with these constraints, like the Parma Polyhedra Library[1].

One important fact to note is that in our model, program variables may point to the same heap objects, because we have assignments of the form `x := y`.

4.3 Describing error conditions

Using polyhedra, we have two possibilities to describe the states that will not fire any assertions later on. Either we keep track of the allowed states and check whether the current heap configuration is in there, or we keep track of the error states and check that no heap configuration lies in the error state.

For our analysis, we will keep track of error conditions.

4.4 Our abstract domain

Because we deal with fields of heaps objects, we have to adapt the notation a little bit. Let $\xi : Vars \rightarrow 2^{Fields}$ denote the function that maps a variable to all the fields that can be associated with it.

The constraints of the polyhedron are on all the fields of all the variables in the current program (note that this will be a very sparse matrix).

Now we can denote the domain of one constraint: First, let's denote the total number of fields over which we can have constraints as $n = |\{f | f \in \xi(x) \wedge x \in X\}|$. Then the domain of one constraint is $K = \mathbb{R}^{n+1}$.

$$(c, b) \in K \Leftrightarrow \sum_{i=1}^{n} c_i f_i \leq b$$

Thus, if we have n constraints over m fields, our abstract domain is

$$A = \mathbb{R}^{n \times (m+1)}$$

an element would be written as the pair (C, b) where $C \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$.

and the polyhedron that is satisfied by fields f such that (for $(C, b) \in A$)

$$C \cdot f \leq b$$

4.5 Concretization function γ

Let $(C, b) \in A$ be the polyhedron bounding fields f of variables $x \in X$. Then we can define a concretization function $\gamma : A \to C$ that describes all possible heaps that we can construct given that they don't fall into the error polyhedron (C, b).

Let $m : V \to H$ be the function that maps from variable names to heap objects.

$$\gamma((C, b)) = \{m(x) | f \text{ contains } \xi(x) \wedge \neg(C \cdot f \leq b)\}$$

4.6 Lattice ordering $\sqsubseteq$

We want to model the set of heap configurations that may produce an error later on. The polyhedron putting the most constraints on heap configurations is the polyhedron spanning all of $\mathbb{R}^n$. This will be the $\bot$ element of our lattice. Consequently, $\top$ will be the polyhedron that puts no constraints on heap configurations, that represents the empty set $\emptyset$.

The ordering $\sqsubseteq$ of the polyhedron has to satisfy the following condition (for polyhedra P, Q): $P \sqsubseteq Q \Rightarrow \gamma(P) \subseteq \gamma(Q)$. This is the case if the error polyhedron P is a superset of the error polyhedron Q: $P \sqsubseteq Q \Leftrightarrow f \in Q \rightarrow f \in P$.

4.7 Abstract interpretation of instructions

For the abstraction to be useful, we have to correctly model the effect that each instruction has on the polyhedron in the abstract domain. We start our backward analysis by assigning $\bot$ to every program node. This way, we can be sure that we don't get any false positives while the backward analysis is still running.

4.7.1 Exit node

The exit node starts with the polyhedron $\top$: If during concrete execution we landed here, then there will be no constraints for a correct termination of the program.

4.7.2 Assertion statements

Assertions directly describe states that will give an error. Say we have the assertion `assert(x.f > 10)`. Directly from this we can deduce the error state: $x.f \leq 10$. We now have to add this error state to the set of error

states that we have obtained before, i.e. take the union of the two polyhedra. However, the union of two polyhedra might not necessarily be convex (say $x.f \leq 5$ and $x.f \geq 6$). Therefore, we have to calculate the convex hull of the error polyhedra, the smallest convex set containing both polyhedra.

$$[\![assert(d(X) \leq e(X))]\!]_{\mathcal{P}} = convex - hull(P, \neg(d(X) \leq e(X)))$$

Where $d(X), e(X)$ are linear combinations on the current program fields.

Note that `assert(P && Q)` is the same as `assert(P); assert(Q)`. Disjunctions `assert(P || Q)` are also handled easily, due to the fact that their error space is just a conjunction of their respective negations. We can again take the convex hull of the error state before the assertion and $\neg P \wedge \neg Q$.

4.7.3 Assume statements

Assume statements restrict the possible error states: If we have `assume(x.f <= 10)` on an error polyhedron $x.f \geq 5$, then the resulting error polyhedron is $5 \leq x.f \leq 10$. So, if the assume statement can be expressed as polyhedron Q, then

$$[\![assume(Q)]\!]_{\mathcal{P}} = P \cap Q$$

Where $\cap$ is the standard conjunction operator on sets (this works because the conjunction of two convex sets is again a convex set).

If, however, Q is not expressible as a polyhedron, we simply ignore it:

$$[\![assume(*)]\!]_{\mathcal{P}} = P$$

4.7.4 Field assignments

Looking at statements of the form `x.f := l` where `l` is a linear combination of fields of variables, we can see that this just results in replacing $x.f$ by l in the inequalities of the polyhedron:

$$[\![x.f := l]\!]_{\mathcal{P}} = P[x.f \to l]$$

However, as you may have noted, our input language allows for non-linear assignments (like $x.f := y.f \cdot z.g$) as well. If such a non-linear expression $*$ is used, we're out of luck: We have to assume that $x.f$ may take any value. This can be expressed in a polyhedron where the $x.f$ dimension is projected onto the other dimensions.

$$[\![x.f := *]\!]_{\mathcal{P}} = P \text{ where } x.f \text{ can take any value}$$

The same thing is true for user-generated input:

$$[\![x.f := input()]\!]_{\mathcal{P}} = [\![x.f := *]\!]_{\mathcal{P}}$$

4.7.5 Variable assignments

If we have an assignment of the form `x := y`, this means that all the fields of x in the conditions accumulated below will be determined by the fields of y. This is equivalent to a series of linear assignments `x.f := y.f; x.g := y.g ...`

$$[\![x := y]\!]_{\mathcal{P}} = [\![x.f := y.f \ \forall \ f \text{ fields of} x]\!]_{\mathcal{P}}$$

The error polyhedron will not impose any constraints on fields of x.

4.7.6 Object instantiation

An instruction of the form `x := new` Associates a new instance with a variable. This means that all error conditions below this point will deal with another instance not present before this instruction. A first approach would then be to replace this instruction with `x := v` where v is a fresh variable. However, this

- Doesn't work in loops
- Is not very precise

On closer inspection, we assume that in each instance all fields will have a default value. So the instruction is abstracted with

$$[\![x := new]\!]_{\mathcal{P}} = [\![x.f := 0 \ \forall \ f \text{ field of } x]\!]_{\mathcal{P}}$$

4.7.7 Nondeterministic branching

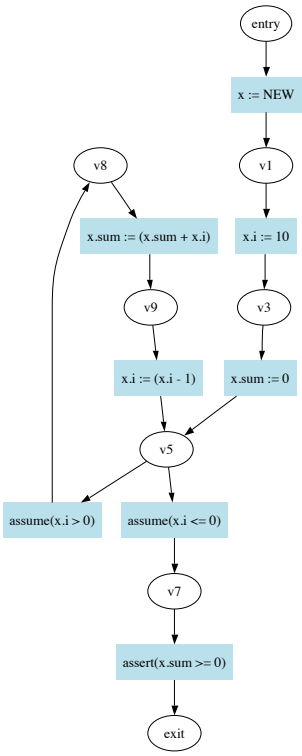
Some of the nodes contain multiple successors, which means that at this point we'll have to collect the error polyhedra from the different branches. For this, we'll have to define the $\sqcup$ (join) operation. We define

$$P \sqcup Q \Leftrightarrow convex - hull(P \cup Q)$$

This follows quite intuitively from the definition: If P denotes the error configurations in one branch and Q denotes the error configurations in the other branch, then the node has to contain all the error configurations from both branches.

4.7.8 Loops and widening

Consider the following program:



Our analysis will find the following error states

v7 $x.sum < 0$

v5 $x.sum < 0 \wedge x.i \leq 0$

v9 $x.sum < 0 \wedge x.i \leq 1$

v8 $x.sum + x.i < 0 \wedge x.i \leq 1$

v5 $convex - hull((x.sum < 0 \wedge x.i \leq 0), (x.sum + x.i < 0 \wedge x.i \leq 1 \wedge x.i > 0)) \equiv x.sum < 0 \wedge x.sum + x.i < 0 \wedge x.i \leq 1$

...

v5 $x.sum < 0 \wedge x.sum + 2 * x.i < 1 \wedge x.i \leq 2$

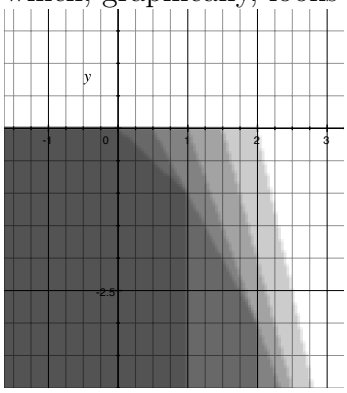
...

v5 $x.sum < 0 \wedge x.sum + 3 * x.i < 3 \wedge x.i \leq 3$

Or, in general, after n iterations we will obtain the polyhedron

$$x.sum < 0 \wedge x.sum + n \cdot x.i < \frac{n \cdot (n-1)}{2} \wedge x.i \leq n$$

which, graphically, looks as follows:



If we would just continue taking the convex hull at every intersection, we will never converge to the final polyhedron (note that the polyhedron converges to $x.sum < 0$). So we need a widening operator.

4.7.9 Standard widening

Say after iteration n we obtained polyhedron S_n and after iteration $n+1$, we had S_{n+1}. Then we can use the standard widening first introduced by Cousot and Halbwachs[2]:

$S_n \nabla S_{n+1}$ is the linear restraints of S_n verified by every element of S_{n+1}. In our example case,

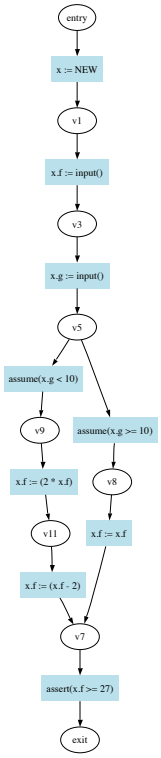
$$S_n \equiv x.sum < 0 \wedge x.sum + n \cdot x.i < \frac{n \cdot (n-1)}{2} \wedge x.i \leq n$$

$$S_{n+1} \equiv x.sum < 0 \wedge x.sum + (n+1) \cdot x.i < \frac{(n+1) \cdot n}{2} \wedge x.i \leq n$$

Note that the only commonality is $S_n \nabla S_{n+1} \equiv x.sum < 0$ this is both bigger than S_n and bigger than S_{n+1}.

4.8 Limitations

Consider the following program:



Our backward analysis proceeds as follows:

v7	$x.f < 27$
v11	$x.f < 29$
v8	$x.f < 27$
v9	$2 \cdot x.f < 29$
v5	$convex - hull(x.g < 10 \wedge 2 \cdot x.f < 29, x.g \geq 10 \wedge x.f < 27) \equiv x.f < 27$

So we lose everything associated with only one branch.

5 Combining the analyses

When we were dealing with backward analysis, we had to restrict ourselves to a very simple representation of the error state. Because of loops, widening had to be possible, and widening only works reliably on simple abstractions (it is, of course possible, to infer loop invariants and thus not losing precision).

But polyhedra had multiple problems. For instance, we could not deal with non-linear assignments:

$$x.f := y.g/z.h$$

Here, we had to approximate by saying that $x.f$ could take any value, which in turn meant that if the error state depended on $x.f$, we would land in it.

But now suppose we could use the information from forward constant propagation to find out that for all possible heap objects that z points to, its value h is c. In this case, we know already that the assignment will be

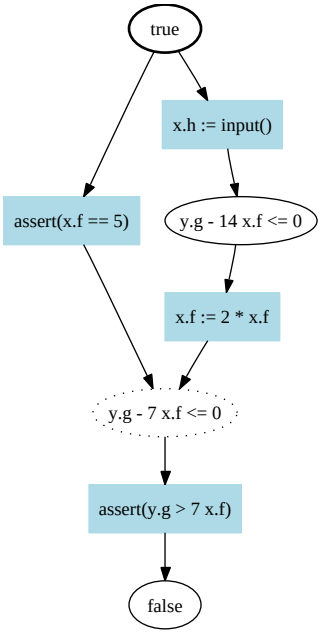
$$x.f := (1/c) * y.g$$

...which in turn means that we can say a lot more about the error polyhedron.

5.1 Forward-skipping

Most of the time, the loss of precision of the backward analysis will be too much that we are able to prove absence of errors at the current program point. This is because for every additional assertion, we will have to build the convex hull with the error states after that. So it might be that our heap configuration lands us in an error state at the current program point, but not later on.

Suppose we are in the bold state at the moment. The current error condition is written on the state.



During our backward analysis, we encountered the assertion $x.f = 5$. As we propagate error states, our rules demand that we obtain the convex hull of $x.f \neq 5 \equiv x.f < 5 \wedge x.f > 5$ and the current error condition. But alas the convex hull of the two half-spaces $x.f < 5$ and $x.f > 5$ is just $x.f < \infty$, $x.f$ can take on any value.

One correct approach would be to integrate the results of constant propagation like we did for non-linear assignments: Then we could replace the `assert(x.f == 5)` with `assert(true)` or `assert(false)`, depending on the results of constant propagation. This would refine the error condition and we would get something more precise for our current state.

However, note that assertions do not change the heap configuration already calculated with forward analysis. So if we find a way to advance to the dotted state and are then able to see that the heap configuration there does not land in the error condition, then we can also guarantee that we do not run into errors.

```
S = new Queue
S += pc
enough = false
while (!S.isEmpty and !enough) {
  s = S.pop
  if (s is an assertion) {
    if (!evaluate(s)) enough = true // The assertion did not hold
  }
}
```

```
Evaluate the error conditions on all states in S with the constant propagation
information. If at no point we landed in an error condition, then we have
proven that the program will not fail anymore.
```

5.2 Evaluating the error state

When we have run the backward analysis up to convergence, we know that we will be error-free if our current heap configuration at this program point is not in an error state. We can check this as follows:

For each variable/field combination used in the error polyhedron, insert the concrete value from the current heap. If the resulting set of equations is not satisfiable, then the current heap is not in the error state.

6 Problems with the polygon approach

While over-approximating abstract error states with polygons allows us to guarantee that some programs will have an error free run, some properties cannot be easily approximated. One often-used property is $x.f \neq 0$, which can be easily approximated with the error state $x.f \leq 0 \wedge x.f \geq 0$.

However, even the property $x.f = y.f$ cannot be expressed, as the error state $x.f > y.f \vee x.f < y.f$ is not convex.

Changing our domain to represent acceptable states with polyhedra is not much of a solution because then we can't handle inequalities.

References

[1] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella, *The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*, Sci. Comput. Program. **72** (2008), no. 1-2, 3–21.

[2] P. Cousot and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Tucson, Arizona), ACM Press, New York, NY, 1978, pp. 84–97.