The ScalaFix equation solver

Gianluca Amato[0000-0002-6214-5198] and Francesca Scozzari[0000-0002-2105-4855]

Lab. of Computational Logic and AI
Department of Economic Studies
University of Chieti–Pescara, Pescara, Italy
`{gianluca.amato,francesca.scozzari}@unich.it`

Abstract. We present ScalaFix, a modular library for solving equation systems by iterative methods. ScalaFix implements several solvers, involving iteration strategies from plain Kleene's iteration to more complex ones based on a hierarchical ordering of the unknowns. It works with finite and infinite equation systems and supports widening, narrowing and warrowing operators. It also allows intertwining ascending and descending chains and other advanced techniques such as localized widening.

Keywords: Static analysis · Equation systems · Iterative methods · Widening · Narrowing

1 Introduction

One of the most common approaches for performing static analysis of software, used for both simple data-flow analysis and more complex analysis based on abstract interpretation, is to setup a set of equations over some partially ordered set. The solutions of this equation system form the result of the analysis.

These equation systems are generally solved by iterative methods, based on some variant of the Knaster–Tarski theorem. This is immediate when the partial order on the values of the unknowns has a small finite height, but becomes difficult when the height is large or, worse, the partial order does not satisfy the ascending chain condition. In this case, some way of accelerating iterations is needed, such as a *widening*/*narrowing* [13] or *warrowing* [8].

The ScalaFix library strives to be a general solver for these kind of equation system, in the spirit of modularization of static analyzers presented in [19]. It implements several iterative algorithms for solving equations (both with a finite or infinite number of unknowns) and it has a convenient interface which is designed for the Scala programming language. Scala combines functional and object-oriented programming in a single high-level language which runs on the Java Virtual Machine (JVM). The library may also be used, with some difficulties, from other languages which run on the JVM, such as Java itself. A better interface for other languages is planned for a later version. The source code of ScalaFix is available on https://github.com/jandom-devel/ScalaFix (in this paper we present the release 0.10), while the compiled code is on the Sonatype OSSRH

(OSS Repository Hosting) https://oss.sonatype.org/ with group `it.unich.scalafix` and artifact `scalafix`.

In this paper we present the structure of the ScalaFix library and we show some examples of its use. The main application target for such a library is to be a backend for a static analyzer (it is currently in use by the Jandom static analyzer [2]).

In all the code fragments appearing in this paper we assume the following import statements:

```
import it.unich.scalafix.{finite, infinite, *}
import it.unich.scalafix.finite.*
import it.unich.scalafix.graphs.*
import it.unich.scalafix.highlevel.*
import it.unich.scalafix.utils.Relation
```

In many examples we will also use the PPL (Parma Polyhedra Library) [10] trough the JPPL bindings. These are simpler and more natural to use than the default Java bindings provided by the PPL. The source code for JPPL is available on https://github.com/jandom-devel/JPPL, while the compiled code is on the Sonatype repository https://s01.oss.sonatype.org/ with group `it.unich.jppl` and artifact `jppl`.

All the examples in this paper are available with full code in the GitHub repository https://github.com/jandom-devel/ScalaFixExamples.

2 Equation systems

The main concept of ScalaFix is the *equation system*. It comes in two flavors: either with a finite number of unknowns or with a possibly infinite number of unknowns. The main difference between the two flavors is that, in the first case, we are generally interested in solving the system for all the unknowns, while in the second case we are only interested in solving for a single unknown.

Each equation system is characterized by a type `U` for the unknowns and a type `V` for the values assumed by the unknowns. As *assignment* is a function from unknowns to values. The different solvers of ScalaFix take an assignment as the input, perform several iterative steps, and produce a new assignment as the solution of the equation system. The *body* of an equation system is the cornerstone of all the iterative algorithms: it takes an initial assignment and returns a new assignment obtained by computing all the right hand sides of the equation system. In the Scala language, we have:

```
type Assignment[-U, +V] = U => V
type Body[U, V] = Assignment[U, V] => Assignment[U, V]
```

where `[-U, +V]` means that the assignment type is covariant in `V` and contravariant in `U`.

It is important not to be misled by the type of `Body`. Given the variables `body: Body[U, V]` and `rho: Assignment[U, V]`, we have that `body(rho)` is a function, hence no real computation starts until this is applied to a specific unknown `u: U`, as in `body(rho)(u)`.

2.1 Infinite equation systems

For example, consider the following equations defining the Fibonacci sequence:

$$x_0 = x_1 = 1 \qquad\qquad x_{i+2} = x_i + x_{i+1}$$

This may be encoded in ScalaFix as follows:

```
val body: Body[Int, BigInt] =
  (rho: Assignment[Int, BigInt]) =>
    (u: Int) =>
      if u <= 1 then 1 else rho(u-1) + rho(u-2)
```

Note that, although many type declarations might be avoided thanks to the Scala type inference, we have decided here to be more verbose since we think it is helpful to the reader.

The body must be packed into an `EquationSystem` before being handed over to a solver:

```
val eqs = EquationSystem(body)
```

Since the number of unknowns is infinite, we use the `infinite.WorkListSolver` for computing a solution. The worklist solver needs three parameters: an equation system, an initial assignment and the set of the unknowns for which we want to get a partial solution. For example, if we want to known the sixth Fibonacci number we use:

```
infinite.WorkListSolver(eqs)(Assignment(1), Set(6))
```

where `Assignment(1)` is the assignment which maps every unknown to 1 and `Set(6)` is the singleton set $\{6\}$.

The output is an assignment which maps 6 to the 6th Fibonacci number. Morevoer, since in order to determine x_6 we also need to solve for the unknowns from x_0 to x_5, the resulting assignment also contains the values for these unknowns:

```
[ 0 -> 1, 1 -> 1, 2 -> 2, 3 -> 3, 4 -> 5, 5 -> 8, 6 -> 13 ]
```

Another solver for infinite equation systems implemented in ScalaFix is the `PriorityWorkListSolver`, where the unknown to be updated is chosen, among those in the worklist, according to priorities which are dynamically generated.

In the general case there is no guarantee of convergence: this should be assured from the specific theory used to analyze the equation system under consideration.

2.2 Finite equation systems

The solvers in the `infinite` package work for equation systems with a possibly infinite number of unknowns. If the number of unknowns is finite, it is possible to use a `FiniteEquationSystem` instead, which allows to use different solvers specifically tailored for this case.

A finite equation system is characterized, besides its body, also by:

– the set of all the unknowns;
– the *influence* relation between the unknowns;
– a subset of all the unknowns, called the *input unknowns*.

While the set of all the unknowns is an obvious information, the other two parameters deserve an explanation. The influence relation determines the dependencies between unknowns. If the unknown x is used in the right hand side of the equation defining y, then we say that x influences y. When x is recomputed and its value changes, all the unknowns influenced by x should be recomputed as well.

Note that not all the solvers actually need the influence relation. For example, it is not used by `KleeneSolver`, which performs a parallel update of all the unknowns, and by `RoundRobinSolver`, which repeatedly updates all the unknowns one at a time. On the contrary, it is used by those solvers which avoid to recompute an unknown when it is not strictly necessary, such as `WorkListSolver`. In the case of infinite equation systems, the influence relation is computed dynamically during the evaluation of the body when needed, while for finite equation systems we require the influence relation to be provided statically.

The set of input unknowns is used to compute a depth-first ordering of the unknowns in the equation system. This allows to determine the set of unknowns where widening should be applied to ensure convergence and the default hierarchical ordering [11] for the `HierachicalOrderingSolver`.

We may turn the Fibonacci example into a finite equation system by restricting the number of unknowns, as follows:

```
val eqs = FiniteEquationSystem(
            body,
            infl = Relation( (i: Int) => Set(i-1, i-2) ),
            unknowns = 0 to 10,
            inputUnknowns = Set(0, 1) )
```

Then we may solve the equation system, for example with:

```
finite.WorkListSolver(eqs)(Assignment(1))
```

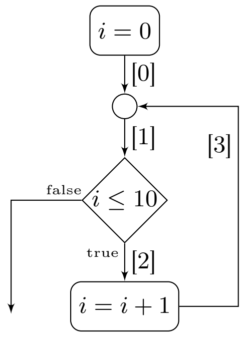
Here we do not need to specify the set of wanted unknowns, since we assume we are interested in solving the entire equation system and find the fixpoint for all the unknowns.

2.3 A use case for static analysis

We show a use case involving the Parma Polyhedra Library trough the Java binding provided by JPPL. Consider the example program `loop` and its corresponding equation system over the interval domain in Figure 1. We first define the body of the equation system using the interval abstract domain `DoubleBox` from PPL as follows:

```
val body = (rho: Int => DoubleBox) => {
  case 0 => DoubleBox.from(/* constraint system {i=0} */)
```

```
    i = 0
[0] while [1] (i<=10) {
[2]   i = i+1
[3] }
```



$$\begin{aligned} x_0 &= [0, 0] \\ x_1 &= x_0 \vee x_3 \\ x_2 &= x_1 \wedge [-\infty, 10] \\ x_3 &= x_2 + [1, 1] \end{aligned}$$

Fig. 1. The example program `loop`. The symbols $\vee$, $\wedge$ and $+$ are respectively the lub, the glb and the sum on the domain of intervals.

```
    case 1 => rho(0).clone().upperBound(rho(3))
    case 2 => rho(1).clone().refineWith(/* constraint i<=10 */)
    case 3 => rho(2).clone().affineImage(0, /* expression i+1 */)
  }
```

We can now construct a finite equation system as follows:

```
val eqs = FiniteEquationSystem[Int, DoubleBox](
    body,
    inputUnknowns = Set(0),
    unknowns = 0 to 3,
    infl = Relation(0 -> 1, 1 -> 2, 2 -> 3, 3 -> 1) )
```

where `unknowns` correspond to the program points 0, 1, 2, 3 in Figure 1 and `infl` defines the dependency relations between the equations. For instance, `0 -> 1` means that any change in the value of x_0 requires recomputation of x_1. We can now solve the equation system with:

```
finite.WorkListSolver(eqs)(Assignment(DoubleBox.empty(1)))
```

whose solution is:

```
[0 -> i in 0, 1 -> i in [0, 11], 2 -> i in [0, 10], 3 -> i in [1, 11]]
```

where `2 -> i in [0, 10]` means that in the program point 2 the value of i is in the interval $[0, 10]$.

2.4 Infinite equation systems and static analysis

While finite equation systems are well-suited for intra-procedural analysis, infinite equation systems may be used for inter-procedural analysis, by including an abstraction of the call-stack as part of the unknowns.

Consider, for example, the following code:

```
function incr(a) {
[1]   b = a+1
[2]   return b
[3]
}

    i = j = 0
[4] j = incr(i)
[5] i = incr(j)
[6]
```

A possible approach for the analysis of this program consists in defining an equation system whose unknowns are pairs (p, c) where p is a program point and c is an abstraction of the call-stack, such as (but not limited to) an abstract representation of the values of the formal parameters.

Assuming to work with the interval domain, this is an excerpt of the equation system which describes the `incr` function:

```
val body: Body[(Int, DoubleBox), DoubleBox] = (rho) =>
  case (1, c) => c.clone().addSpaceDimensionAndEmbed(1)
  case (2, c) => rho((1, c)).clone()
                           .affineImage(1, /* expression a+1 */)
  case (3, c) => rho((2, c))
```

When the function `incr` is called in the context c, the value of the variables at program point 1 is obtained by enlarging the input, provided in c, with a new unconstrained dimension representing the variable b. The equation for the return statement, i.e., the unknown $(3, c)$, is a no-op: in the general case, we might decide to remove those dimensions corresponding to the local variables not returned by the function.

The following are the equations for the main program:

```
  case (4, c) => DoubleBox.from(/* constraints {i=j=0} */)
  case (5, c) =>
    val call_context = rho((4, c)).clone()
                                  .removeSpaceDimensions(Array(1))
    val return_context = rho((3, call_context))
    val result = /* combine rho((4, c)) and return_context */
    result
  case (6, c) =>
    /* similar to code for (5, c) */
```

The interesting point is the equation for the program point $(5, c)$. Here we:

1. determine the abstract calling context by projecting the abstract value of the program point 4 on the actual parameters of the function call (variable i in this case);
2. take the abstract return context of the function `incr`, when invoked with the previously computed calling context;

3. combine the information at program point 4 and the return context to get the final best possible approximation of the value of variables at program point 5.

A theoretical discussion on the last step, as well as on alternative abstractions of the call-stack, is available in [21].

The reason why infinite equation systems are useful for inter-procedural analysis is that we cannot know in advance the contexts we will use for evaluating the `incr` function. In this example ScalaFix uses the intervals $[0, 0]$ and $[1, 1]$. Note that, in more complex cases, some kind of widening operator should be applied on calling contexts to avoid generating an infinite number of them.

3 Widening, narrowing and warrowing

ScalaFix supports the use of widenings, narrowings [14] and warrowings [8]. These operators are commonly used to combine the values of the last two iterations into a new value, in order to accelerate or ensure the convergence. In this paper, and in the ScalaFix jargon, they are generally called *combos*. Combos are implemented at the level of an equation system, and therefore work with every fixpoint solver. Mathematically, a combo over a set V is a binary function $\Box : V \times V \to V$. In ScalaFix we have that:

```
type Combo[V] = (V, V) => V
```

Applying a combo $\Box$ to an unknown x_i means replacing the equation $x_i = e$ with $x_i = x_i \Box e$. Typically, a combo is applied to a selection of unknowns, generally the loop heads in the graph generated by the unknowns and their influence relation. Potentially, we might want to use different combos for different unknowns. Therefore, when using combos in ScalaFix, we need to specify a `ComboAssignment`, i.e., a partial function which maps each unknown to the combo we want to use for it (if any). Continuing the example in Section 2.3, we may define a combo using the standard widening for intervals [12]:

```
val widening = Combo[DoubleBox]( (x: DoubleBox, y: DoubleBox) =>
                   y.clone().upperBound(x).widening(x) )
val comboAssignment = ComboAssignment(widening).restrict(Set(1))
```

where `restrict(Set(1))` means that we apply the widening to the unknown 1 only. We now equip the equation system with the widening:

```
val eqsWithWidening = eqs.withCombos(comboAssignment)
```

The equation system can be solved as before:

```
finite.WorkListSolver(eqsWithWidening)(Assignment(DoubleBox.empty(1)))
```

ScalaFix also implements general techniques enhancing widenings and narrowings such as delayed widening.

3.1 Automatic determination of combo points

Instead of manually specifying the set of unknowns where combos should be applied, we may let ScalaFix determine this set automatically. Each finite equation system induces a dependency graph whose nodes are the unknowns and such that there is an edge (x, y) iff x influences y. We may build a depth-first ordering of this graph using

```
val ordering = DFOrdering(eqs)
```

whose result for the example program `loop` is:

```
UnknownOrdering( 0 (1) 2 3 )
```

Here the parenthesis denotes loop head nodes, i.e., nodes which are the target of retreating edges. In order to ensure convergence, it is enough to apply widenings to these nodes. This may be done with the `restrict` method used above, using the graph ordering as a parameter:

```
val comboAssignment = ComboAssignment(widening).restrict(ordering)
```

Then, everything proceeds as in the previous example.

4 Equation systems based on hyper-graphs

In the equation system shown above, the right-hand side of equations are black boxes. This is generally fine, but in some cases exposing some structure allows optimizations which are not possible otherwise. This is especially true for unknowns such as x_1 in Figure 1 which correspond to join nodes of a flow chart.

ScalaFix allows to define a body for an equation system in a way that makes manifest the individual contributions of the edges of the flow chart. Consider again the equations in Figure 1. For the sake of clarity, in Figure 2 we depict the control-flow graph of the program. Note that the edge `i=0` has no source: this is fine since ScalaFix supports hyper-graphs, where each edge may have many (possibly none) sources and a single target. Hyper-graphs are needed for inter-procedural analysis [19]. Edges `enter` and `loop` correspond to the two edges entering the join node in Figure 1, i.e., to the contributions x_0 and x_3 in the equation $x_1 = x_0 \vee x_3$.

We need to associate to each edge an action, i.e., a function that takes an assignment and returns the contribution of that edge to the new value of the target unknown.

```
type EdgeAction[U, V, E] = Assignment[U, V] => E => V
```

For our example equation system we have:

```
val edgeAction = (rho: Assignment[Int, DoubleBox]) => {
  case "i=0"   => DoubleBox.from(/* constraint system {i=0} */)
  case "enter" => rho(0)
```

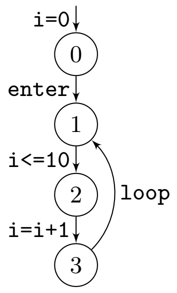


Fig. 2. The graph corresponding to the equation system in Figure 1.

```
    case "i<=10" => rho(1).clone().refineWith(/* constraint i<=10 */)
    case "i=i+1" => rho(2).clone().affineImage(0, /* expression i+1 */)
    case "loop" => rho(3)
  }
```

The actions for the edges should be packed together with fields describing the structure of the graph into a `GraphBody`:

```
  val graphBody = GraphBody[Int, P, String](
    sources = Relation(
      "enter" -> 0, "i<=10" -> 1, "i=i+1" -> 2, "loop" -> 3),
    target = Map(
      "i=0" -> 0, "enter" -> 1, "i<=10" -> 2, "i=i+1" -> 3, "loop" -> 1),
    ingoing = Relation(
      0 -> "i=0",  1 -> "enter", 1 -> "loop", 2 -> "i<=10", 3 -> "i=i+1"),
    outgoing = Relation(
      0 -> "enter", 1 -> "i<=10", 2 -> "i=i+1", 3 -> "loop"),
    edgeAction = edgeAction,
    combiner = (x, y) => x.clone().upperBound(y),
    unknowns = 0 to 3 )
```

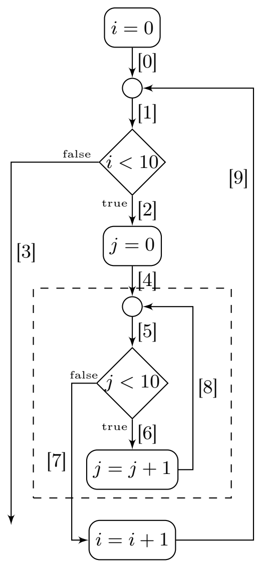
The body is automatically reconstructed in ScalaFix by combining all the contributions from the incoming edges with the specified operation `combiner`, which in our example is simply the upper bound operator of the abstract domain. Finally, the body is used to build a graph-based equation system:

```
  val eqs = GraphEquationSystem(
    initialGraph = graphBody,
    inputUnknowns = Set(0) )
```

Since a `GraphEquationSystem` is a subclass of `FiniteEquationSystem`, we may use `eqs` exactly as the equation systems in the previous sections.

Note that the way we provide to `GraphBody` the structure of the graph is not particularly elegant: there is a lot of redundancy among the parameters `sources`, `target`, `ingoing` and `outgoing`. However, ScalaFix has been principally designed to be used as a backend for a static analyzer. In this context, it is likely

```
    i = 0
[0] while [1] (i<10){
[2]   j = 0
[4]   while [5] (j<10)
[6]     j = j+1  [8]
[7]   i = i+1    [9]
    } [3]
```



$$
\begin{aligned}
x_0 &= [0,0] \times [-\infty,\infty] \\
x_1 &= x_0 \vee x_9 \\
x_2 &= x_1 \wedge ([-\infty,9] \times [-\infty,\infty]) \\
x_3 &= x_1 \wedge ([10,\infty] \times [-\infty,\infty]) \\
x_4 &= \mathit{first}(x_2) \times [0,0] \\
x_5 &= x_4 \vee x_8 \\
x_6 &= x_5 \wedge ([-\infty,\infty] \times [-\infty,9]) \\
x_7 &= x_5 \wedge ([-\infty,\infty] \times [10,\infty]) \\
x_8 &= x_6 + ([0,0] \times [1,1]) \\
x_9 &= x_7 + ([1,1] \times [0,0])
\end{aligned}
$$

Fig. 3. The example program `nested`.

that the analyzer has already built the control-flow graph internally. Since the four parameters above are just functions from edges (or nodes) to set of nodes (or edges), it is easy for a static analyzer to build a very thin layer providing these parameters.

The ScalaFix library also provide a different API for building graphs (the `GraphBodyBuilder` class) which is easier to use for simple experiments but is not described in this paper.

4.1 Localized widening

The definition of an equation system based on hyper-graphs allows us to use localized widening [7]. Consider the program `nested` in Figure 3 and the corresponding system of equations. Let `graphBody` be the description of the graph in Figure 3, as depicted in Figure 4. We can build, as in the previous section, a graph equation system as follows:

```
val eqs = GraphEquationSystem(
  initialGraph = graphBody,
  inputUnknowns = Set(0) )
```

and define the widening:

```
val widening = Combo[DoubleBox]((x: DoubleBox, y: DoubleBox) =>
                 y.clone().upperBound(x).widening(x))
```

Now using `DFOrdering` we can recover the depth-first ordering of the set of unknowns:

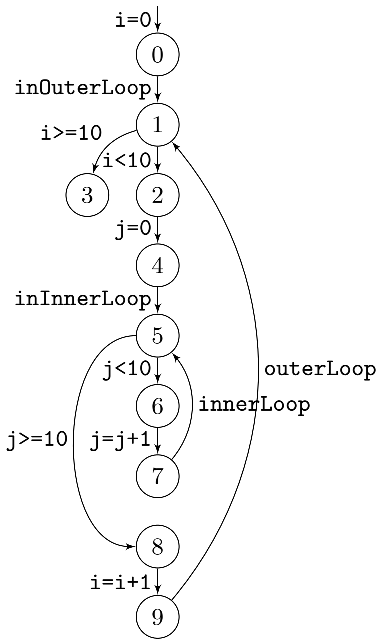


Fig. 4. The graph corresponding to the equation system in Figure 3.

```
val ordering = DFOrdering(eqs)
```

which is `( 0 (1) 3 2 4 (5) 8 9 6 7 )`, where `(1)` and `(5)` are the loop head nodes. We can apply localized widening to these nodes as follows:

```
val widenings = ComboAssignment(widening).restrict(ordering)
val eqsWithWidening = eqs.withLocalizedCombos(widenings, ordering)
val solutionAscending =
      WorkListSolver(eqsWithWidening)(Assignment(DoubleBox.empty(2)))
```

where the last line computes the solution for the ascending chain. We can now start a descending phase using the narrowing defined in [12]:

```
val narrowing = Combo[DoubleBox]((x: DoubleBox, y: DoubleBox) =>
                  y.clone().intersection(x).CC76Narrowing(x))
val narrowings = ComboAssignment(narrowing).restrict(ordering)
val eqsWithNarrowing = eqs.withCombos(narrowings)
WorkListSolver(eqsWithNarrowing)(solutionAscending)
```

In the solution for the program point `3` we have that `i in [10, 11)`, which cannot be computed without the localized widening.

5 A high-level interface

The interface shown above, where the user builds an equation system, decides where to apply widening/narrowing and calls the solver with appropriate parameters, is rather low-level. For example, if one wants to solve an equation system using the classical approach based on an ascending chain with widening followed by a descending chain with narrowing, this procedure must be repeated for both phases, as done in the previous section.

Albeit this allows an extreme flexibility, if we just want to solve an equation system following a standard approach, ScalaFix provides a high-level API which simplifies this task. It is enough to call the generic `FiniteFixpointSolver` with a bunch of parameters which specify how we want to solve the equation system. For example, the analysis shown in Section 4.1 may be implemented more easily as follows:

```
val params = Parameters[Int, DoubleBox](
  solver = Solver.WorkListSolver,
  start = Assignment(DoubleBox.empty(2)),
  comboLocation = ComboLocation.Loop,
  comboScope = ComboScope.Localized,
  comboStrategy = ComboStrategy.TwoPhases,
  restartStrategy = RestartStrategy.None,
  widenings = ComboAssignment(widening),
  narrowings = ComboAssignment(narrowing) )
FiniteFixpointSolver(eqs, params)
```

The possible choices for the above parameters are:

- `solver`: one of the following fixpoint solvers:
  - `KleeneSolver`: updates all the unknowns in parallel;
  - `RoundRobinSolver`: updates one unknown at a time, following a fixed ordering;
  - `WorkListSolver`: updates one unknown at a time, taken from a queue containing only the unknowns which might produce a different result w.r.t. the previous iteration;
  - `PriorityWorkListSolver`: it is similar to the `WorkListSolver`, but the order in which unknowns are extracted from the queue depends on an ordering of the unknowns;
  - `HierarchicalOrderingSolver`: updates the unknowns following a hierarchical ordering (see [11]).

  For the `PriorityWorkListSolver` and `HierarchicalOrderingSolver`, the ordering is based on the depth first traversal of the equation system.
- `comboLocation`: `None` does not use combos; `All` puts combos at each unknown; `Loop` places combos only at loop heads (which are automatically computed).
- `comboScope`: `Standard` or `Localized`, for standard or localized widening respectively.

– `comboStrategy`: `OnlyWidening` uses widening operators with no descending phase; `TwoPhases` uses the standard two phases widening/narrowing approach; `Warrowing` strictly intertwines ascending and descending steps in a single warrowing operator;
– `restartStrategy`: either `None` or `Restarting` for disabling or enabling the restarting policy which replaces part of the current assignment with the initial assignment, in order to improve precision [8] (only useful for the `PriorityWorklistSolver`).

The high level API also needs some extra information on the analysis domain. This may be provided to ScalaFix in the form of a *given instance* (the Scala equivalent of a type class) of the type `Domain`. This instance implicitly provides the partial ordering relation and the upper bound operator for a given type. This is a fragment of the `Domain` instance for `DoubleBox`:

```
given DoubleBoxDomain: Domain[DoubleBox] with
  def lteq(x: DoubleBox, y: DoubleBox): Boolean = y.contains(x)
  def upperBound(x: DoubleBox, y: DoubleBox): DoubleBox =
       x.clone().upperBound(y)
```

6 Performance

In this section we present some benchmarks showing the performance of the ScalaFix library. Obviously, different equation solvers will have different performances, but comparing different methods for solving equation systems is not in the scope of this paper. What we want to show is the overhead which is caused by using the ScalaFix library instead of an ad-hoc equation solver.

6.1 A simple benchmark using the PPL

Consider the equation system E given by the following equations on $\mathcal{P}(\mathbb{Z})$:

$$\begin{aligned} x_0 &= (x_{N-1} \cap \{v \mid v \leq l\}) \cup \{0\} \\ x_{i+1} &= \{v+1 \mid v \in x_i\} \end{aligned} \tag{1}$$

We solve E with $N = 100$ and $l = 2{,}000$ using the following methods:

1. an ad-hoc implementation of the round-robin solver, using arrays as the data structure for assignments (`array`);
2. an ad-hoc implementation of the round-robin solver, using hash tables as the data structure for assignments (`hash`);
3. the round-robin solver of ScalaFix (`scalafix`).

For each method, we used both the `DoubleBox` and `CPolyedron` domains of the PPL, with or without widening at each unknown. In ScalaFix widenings are added to E using the `.withCombos` method, while in the custom solvers they are inlined inside the solvers.

Benchmarks	**array**	**hash**	**scalafix**
Box without combos	54.364 ± 6.725	55.180 ± 6.318	54.941 ± 7.395
Box with combos	246.072 ± 36.047	261.492 ± 36.443	261.707 ± 35.318
Polyhedra without combos	14.334 ± 2.232	14.994 ± 1.759	15.081 ± 1.515
Polyhedra with combos	85.590 ± 17.383	90.507 ± 13.428	85.638 ± 16.541
Reaching definitions	15946.052 ± 64.141	15298.415 ± 134.841	15301.827 ± 59.145

Table 1. Benchmarks results (operations/s) with 99% confidence intervals.

Benchmarking programs running on the JVM is not an easy task, since a lot of factors may impact the execution speed, such as just in time compilation and garbage collection. We have used the JMH (Java Microbenchmark Harness) to perform the benchmarks, using 5 forks, each fork composed of 5 iterations for warming up the JVM and 5 iterations for collecting the results. On top of this, we have tried to reduce the effect of automatic CPU performance scaling by disabling Turbo Boost and setting a fixed clock for the CPU, low enough not to overheat the processor. In particular, the results have been obtained on a Intel Core i2500K clocked at 1.6GHz.

The results are shown in Table 1, and are expressed in operations per second (i.e., the number of times the equation system is solved per second) with a 99% confidence interval.

The benchmarks show that the difference between the three solvers is negligible, since the cost of executing the `DoubleBox` and `CPolyhedron` operations is much larger than the overhead of the fixpoint solvers.

6.2 Reaching definitions

The second benchmark contains different implementations of an equation system for reaching definition analysis of a three-address code program from [1, p. 626], whose code is in Figure 5. As before, we have executed the benchmark comparing the ScalaFix solver to an ad-hoc implementation of the round-robin solver, using arrays and hash tables. The results show that even in this case the difference between the solvers is negligible.

The experiments suggest that the overhead of using ScalaFix is very limited, almost zero.

7 Related work

Most available static analyzers, both for industrial or academic applications, implement their custom procedure for solving equation systems. We believe that the use of ScalaFix could help developers in experimenting with different and state-of-the-art solvers. Also, they could contribute, by implementing new techniques that would be immediately reusable by the community. Moreover, the developers would benefit from all the experiments and development efforts behind the library. Actually, one of the major difficulty in the development of

```
i = m-1     [d1]
j = n       [d2]
a = u1      [d3]
do
  i = i+1   [d4]
  j = j-1   [d5]
  if (e1)
    a = u2  [d6]
  else
    i = u3  [d7]
while (e2)
```

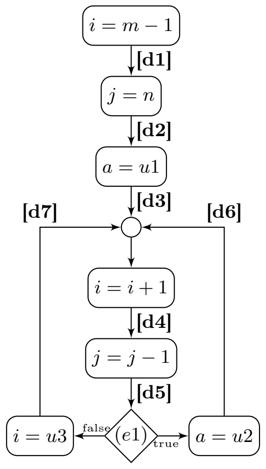


Fig. 5. Reaching definition benchmark.

ScalaFix has been to choose the correct abstractions to put widening, narrowing, warrowing, localized techniques, equation systems, assignments, solvers, etc. . . inside a common API.

To the best of our knowledge, ScalaFix is the only general purpose library for solving equation systems for static analysis which is currently available.

We are aware of only another proposal in the past with the library Fixpoint [20]. This library is unmaintained for more than nine years now and the subversion repository for the source code is not accessible. In general, while Fixpoint and ScalaFix share the same general goal, there are many differences:

- Fixpoint was written in OCaml, while ScalaFix is written in Scala for the Java Virtual Machine.
- The structure of Fixpoint was more monolithic than that of ScalaFix: the `Fixpoint.manager` type encapsulates almost all the information needed to solve an equation system, from the position of widenings to the action of the hyper-edges. In ScalaFix we give different responsibilities to different classes.
- Fixpoint had additional modules implementing some techniques for solving fixpoint equations, namely, guided static analysis [17] and widening with threshold [23]. Implementation of these techniques is a planned improvements for ScalaFix.
- ScalaFix implements many state-of-the-art techniques recently proposed, such as localized widening, warrowing and restarting.
- ScalaFix implements general solvers for infinite equation systems, suitable for the analysis of inter-procedural programs.

Since the source code of Fixpoint is no more available, neither a more detailed comparison nor a performance evaluation has been possible.

Another library for solving fixpoint equations, with a different purpose, is Killdall (https://compcert.org/doc-1.6/html/Kildall.html), written for the Coq proof assistant, and part of the CompCert project [22]. Killdall implements the same algorithm as the `PriorityWorklistSolver` for finite equation systems in ScalaFix, using the depth-first ordering of the equation system for deciding priorities. However, Killdall does not implement any of the additional features of ScalaFix such as combos (Kildall does not have any support for widening or narrowing), infinite equation systems or alternative solvers. But here the goal is to provide a mechanized verification of program analyses, which can be used to equip the CompCert C compiler, being a challenge to implement and reason upon data structures in a purely functional setting such as Coq.

Finally, FPSolve [15] is a library for solving systems of polynomial equations over a semi-ring. While in particular cases it is possible to recast data-flow equations as equations over a semi-ring, this does not hold in general. Therefore the applicability of FPSolve as a general procedure for solving data-flow equations is limited.

8 Conclusion

We have shown some features of the ScalaFix library. There are other features of ScalaFix which are not presented here, such as:

- support for observing the behaviour of the solvers with the listener class `FixpointSolverTracer` which can be used for debugging and computing metrics, and also for fine-tuning the analysis domain using statistical approaches (see for instance [3,6]);
- support for *restarting*: a policy which, under certain conditions, replaces part of the current assignment with the initial assignment, in order to improve precision [8];
- implementation of other equation solvers from the literature, such as solvers based on hierarchical ordering and priority worklists.

ScalaFix is the only general purpose library implementing advanced techniques such as localized widening and restarting. In the near future, we plan to enhance ScalaFix along several directions:

- develop a thin interface layer to make ScalaFix easier to use by other JVM based languages;
- implement more techniques such as guided abstract interpretation [17], lookahead widening [16] or the improved handling of descending chains in [18];
- implement equation systems with side-effects [9] and for different paradigms [4,5].

We have shown in Section 6 that the overhead of using ScalaFix instead of re-implementing an ad-hoc solver is negligible. A big effort has been provided to design the ScalaFix API to be as flexible as possible for the need of very different analyzers, and in the choice of the data structures both for equation systems and graphs to allow the implementation of many speed-up features, depending on the kind of equation systems used.

References

1. Aho, A.V., , Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison Wesley, first edn. (1986)
2. Amato, G., Di Nardo Di Maio, S., Scozzari, F.: Numerical static analysis with Soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. SOAP '13, ACM, New York, NY, USA (2013). DOI: 10.1145/2487568.2487571
3. Amato, G., Parton, M., Scozzari, F.: A tool which mines partial execution traces to improve static analysis. In: Barringer, H., *et al.* (eds.) First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, Lecture Notes in Computer Science, vol. 6418, pp. 475–479. Springer, Berlin Heidelberg (2010). DOI: 10.1007/978-3-642-16612-9
4. Amato, G., Scozzari, F.: Optimality in goal-dependent analysis of sharing. Theory and Practice of Logic Programming **9**(5), 617–689 (Sep 2009). DOI: 10.1017/S1471068409990111
5. Amato, G., Scozzari, F.: Observational completeness on abstract interpretation. Fundamenta Informaticae **106**(2–4), 149–173 (2011). DOI: 10.3233/FI-2011-381
6. Amato, G., Scozzari, F.: Random: R-based analyzer for numerical domains. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7180, pp. 375–382. Springer, Berlin Heidelberg (2012). DOI: 10.1007/978-3-642-28717-6_29
7. Amato, G., Scozzari, F.: Localizing widening and narrowing. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013, Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 25–42. Springer, Berlin Heidelberg (2013). DOI: 10.1007/978-3-642-38856-9_4
8. Amato, G., Scozzari, F., Seidl, H., Apinis, K., Vojdani, V.: Efficiently intertwining widening and narrowing. Science of Computer Programming **120**, 1–24 (May 2016). DOI: 10.1016/j.scico.2015.12.005
9. Apinis, K., Seidl, H., Vojdani, V.: Side-effecting constraint systems: A swiss army knife for program analysis. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 157–172. Springer, Berlin Heidelberg (2012). DOI: 10.1007/978-3-642-35182-2_12
10. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1–2), 3–21 (2008). DOI: 10.1016/j.scico.2007.08.001
11. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Formal Methods in Programming and Their Applications, International Conference Academgorodok, Novosibirsk, Russia June 28 – July 2, 1993 Proceedings, Lecture Notes in Computer Science, vol. 735, pp. 128–141. Springer, Berlin Heidelberg (1993). DOI: 10.1007/BFb0039704
12. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming. pp. 106–130. Dunod, Paris, France (1976)
13. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL

'77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252. ACM Press, New York, NY, USA (Jan 1977). DOI: 10.1145/512950.512973
14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP '92 Leuven, Belgium, August 26–28, 1992, Proceedings, Lecture Notes in Computer Science, vol. 631, pp. 269–295. Springer, Berlin Heidelberg (1992). DOI: 10.1007/3-540-55844-6_142, invited paper
15. Esparza, J., Luttenberger, M., Schlund, M.: FPsolve: A generic solver for fixpoint equations over semirings. In: Holzer, M., Kutrib, M. (eds.) Implementation and Application of Automata. 19th International Conference, CIAA 2014, Giessen, Germany, July 30 – August 2, 2014, Proceedings. Lecture Notes in Computer Science, vol. 1490, pp. 1–15. Springer, Berlin Heidelberg (2014). DOI: 10.1007/978-3-319-08846-4_1
16. Gopan, D., Reps, T.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 452–466. Springer, Berlin Heidelberg (2006). DOI: 10.1007/11817963_41
17. Gopan, D., Reps, T.: Guided static analysis. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007., Lecture Notes in Computer Science, vol. 4634, pp. 349–365. Springer, Berlin Heidelberg (2007). DOI: 10.1007/978-3-540-74061-2_22
18. Halbwachs, N., Henry, J.: When the decreasing sequence fails. In: Miné, A., Schmidt, D. (eds.) Static Analysis, 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7460, pp. 198–213. Springer, Berlin Heidelberg (2012). DOI: 10.1007/978-3-642-33125-1_15
19. Jeannet, B.: Some experience on the software engineering of abstract interpretation tools. Electronic Notes in Theoretical Computer Science **267**(2), 29–42 (2010). DOI: https://doi.org/10.1016/j.entcs.2010.09.016, https://www.sciencedirect.com/science/article/pii/S1571066110001453, proceedings of the Tools for Automatic Program AnalysiS (TAPAS)
20. Jeannet, B.: Fixpoint (2012), http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/
21. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings. vol. 3116, pp. 258–273. Springer, Berlin Heidelberg (2004). DOI: 10.1007/978-3-540-27815-3_22
22. Kästner, D., Leroy, X., Blazy, S., Schommer, B., Schmidt, M., Ferdinand, C.: Closing the gap – the formally verified optimizing compiler CompCert. In: SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium. pp. 163–180. CreateSpace (2017)
23. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6996, pp. 492–502. Springer, Berlin Heidelberg (2011). DOI: 10.1007/978-3-642-24372-1_38