Heuristic Methods for Inductive Invariant Generation in π

Clemens Fruhwirth ⟨clemens@endorphin.org⟩ [*]

May 28, 2008

Abstract

This work present static analysis techniques for the π language. We discuss the distinguishing concepts of π, namely program annotations in First Order Logic, and its ability to automatically check these program annotations (when operating in decidable FOL theories). The process automatically infering FOL annotations by static analysis is problematic because of non-termination issues for all except trivial cases. Interval Analysis and Karr's Analysis are examples of other domains of reasoning that do not exhibit the problematic preciseness of FOL that prevents the static analysis procedure from terminating in the general case.

1 The Language π

Pi(π) is a turning-complete imperative programming language in the syntactic spirit of Java that augments the language with powerful compile-time checked program annotations. π's annotations use First-Order Logic (FOL) to specify behaviour of functions and loops and by utilizing only decidable fragments of FOL theories, the compiler can check these annotations automatically.

The simple π-program LinearSearch demonstrates searching an array a for an element e.

```
bool LinearSearch(int[] a, int e) {
  for @ T
  (int i:= 0: i < |a|; i:=i+ 1) {
    if(a[i]==e) return true;
  }
  return false;
}
```

Clearly, this function returns *true* when the given array contains the element and returns *false* otherwise. The last sentence formulates a semantic insight about the program that no experienced programmer would have trouble concluding. However, it is not resembled by anything found in the source code of this *pi* program. By inspecting the abstract source tree, all we are able to see is a loop containing a return statement guarded by a conditional, and a default return statement. The process of reaching conclusions about the semantics of a function – even if they are so trivial as in this case – require human ingenuity.

In π, the programmer turns that semantic insights into actual source code by annotating function declarations with `@pre` and `@post` assertions expressed in FOL. For both assertions, the formal function parameters are available for reasoning. Additionally, `@post` can use a special variable called rv to reason about the function's return value.

Formulating a precondition for LinearSearch is pretty easy. To work properly, LinearSearch does not require a or e to fulfill certain predicates.[1] Hence, we just require *true* to be valid as precondition (which it trivially is). The following formula captures the semantic insight formulated above, namely that if and only if LinearSearch returns *true* the array a contains the element e.

$$rv \Leftrightarrow \exists j : 0 \leq j < |a| \wedge a[j] = e \tag{1}$$

Type systems have a long history in static analysis and as recent developments have shown, comfortable type inference is possible without much programmer intervention. Is such a comfort also possible for specifications like? The answer for the general case is no. When used with a precise specification domain such as FOL, the turing-completeness of π prevents the inference process from terminating. At the limit it would produce a specification that is complete and correct but this method of reason is operationally infeasible. Human intervention is in general necessary to deliver these limit constructs

[*] This term paper is based on [BM07], Chapter 12 and this paper's findings shall not be confused with novel findings of the author

[1] Except that a is an integer array, while e is an integer. Other languages unify the concept preconditions and types into a single systems.

```
@pre true
@post $rv \leftrightarrow \exists j : 0 \le j < |a| \wedge a[j] = e$
bool LinearSearch(int[] a, int e) {
  for @ true
  (int i:= 0: i < |a|; i:=i+ 1) {
    if(a[i]==e) return true;
  }
  return false;
}
```

Figure 1: LinearSearch annotate with @pre, @post

```
@pre true
@post rv >= 0
int abs(int x) {
  if(x < 0)
    x:=-x;
  // (*)
  return x;
}
```

Figure 2: abs computes the absolute value

by hand and the compiler then takes the responsibility for checking these annotations.

2 Static Analysis: Verification

Verifying programs that contain only boolean values as first-class citizens is easy. We simply check that the postcondition holds for all possible boolean assignment for which the precondition holds. Actually we exhaustively interpreted the program for all possible program states. But as soon as we admit integers, higher-order functions or any other data type with an infinitely large domain, checking postconditions by exhaustive interpretation is infeasible as it implies an infinite amount of work.

Symbolic interpretation is an alternative to that. Instead of working with values, we replace variable content by symbols. The cost of this is that most of the time we are not able to decide whether a condition holds or not. Therefore we must consider both outcomes, and fork the interpretation into two branches, one that assume the guard holds and one that assumes the guard does not hold. As this effectively removes all flow control constructs, the branches inspected only contain assumptions or assignments. To give a simple example, Figure 2 contains a function that computes the absolute value with the help of a conditional.

Instead of checking whether a @post-predicate $p[rv]$ holds after executing `return` e, it is equally effective to check whether $p[e]$ holds right before the return statement. In this case, we have to check $x \ge 0$ at (*) in the source. To verify whether the @pre-condition implies this, we have to consider two cases, namely when $x < 0$ and when $x \nless 0$.

Two branches are created and both have the task to proof that the postcondition is implied by the precondition and the function's statements. These are called verification conditions (VC) and are written with the pre- and postconditions enclosing the statements,

$$\{true\}\texttt{assume } x < 0\texttt{; x:=-x}\{x \ge 0\}$$

and

$$\{true\}\texttt{assume } x \nless 0\{x \ge 0\}$$

The latter branch seems trivial even when we have not formalised the semantic of assume statements. However to reason for the first branch, we need to introduce the concept of predicate transformers to inspect how assumptions and assignments interact. Two transformers are available for this job: the weakest precondition transformer for moving conditions upward in the stream of statements, and the strongest postcondition transformer for moving predicates downward. To ensure that these transformers only generate valid pre- and postconditions, we must define what preconditions and postconditions must fulfill to be valid.

F' is a valid postcondition for F and S, if whenever S is executed on states that fulfill F, the resulting states fulfill F'. More concisely,

$$s \models F \rightarrow \forall s' \in S(s).s' \models F' \qquad (2)$$

for all states s.[2] The most trivial postcondition is *true*. It is too weak as we can not construct any useful implication from *true* as antecedent.

With respect to a statement S and a formula F', a precondition F is valid when for all states that fulfill F, the states also fulfill F' after they were modified by the statement S. This reads very similar to the definition of a postcondition, only from the other perspective. In fact, we can use the same characterization 2. When we use $S^{-1}(s)$ as the set of values that have s as their image in S,

[2]S(s) contains just one element if the language is deterministic, which π is

we can give an alternative characterization for preconditions,

$$s \models F' \to \forall s' \in S^{-1}(s).s' \models F$$

Notice that this characterization is also valid for postconditions.

false is the strongest and most trivial precondition as it valid for all possible F', and S. It is undesirably strong and preconditions get more useful when they get weaker. By focusing on the weakest precondition, we guarantee that we gained as much as possible information about the predecessors states of a formula.

The predicate transformers wp and sp are defined over assumption and assignment statements. The weakest precondition transformer wp is defined for assumptions as.

$$\mathrm{wp}(F, \texttt{assume } c) := c \to F$$

If we know that after passing a guard that enforced c, the predicate F holds, then we know that before the guard $c \to F$ holds, namely F holds in the event that c holds.

$$\mathrm{wp}(F[v], v\texttt{:=}e) := F[e]$$

If a formula $F[v]$ over a variable v holds after the variable received its value from an expression e then the formula must also hold before that statement when we apply it directly the expression e as with $F[e]$. The weakest predicate is defined recursively over a sequence of statements processing the last statement first:

$$\mathrm{wp}(F, S_1 \dots S_k) := \mathrm{wp}(\mathrm{wp}(F, S_k), S_1; \dots; S_{k_1})$$

The strongest postconditions for FOL formulas are derived as follows:

$$\mathrm{sp}(F, \texttt{assume } c) := c \wedge F$$

Moving over assumption statements practically means moving into one branch of a conditional. To reflect that in the formula, we simply add the knowledge that the conditional's guard holds.

$$\mathrm{sp}(F[v], v := e) := \exists v^0 : v = e[v^0] \wedge F[v^0]$$

If formulas holds for a value and this value gets transformed by the expression e, then the formula must hold for at least one inverses of the new value under the expression e. For formulas over expressions that have unique inverses in the underlying theory, the existential quantifier can be removed and the whole formula rewritten to $F[e^{-1}(v)]$. For instance, $\mathrm{sp}(F[x], x := x + 1) = F[x - 1] \wedge x = x_0 + 1$.

For a sequence of statements, the strongest postcondition is defined recursively starting with the first statement,

$$\mathrm{sp}(F, S_1 \dots S_k) := \mathrm{sp}(\mathrm{sp}(F, S_1), S_2; \dots; S_k)$$

Now we are fit to return to the verification task $\{true\}\texttt{assume } x < 0\texttt{;x:=-x}\{x \geq 0\}$. Using wp and sp we can either move $x \geq 0$ backwards with the help of wp and check whether

$$true \to \mathrm{wp}(\mathrm{wp}(x \geq 0, x := -x), \texttt{assume } x < 0)$$

holds, or we can move *true* towards $x \geq 0$ with the help of sp and check

$$\mathrm{sp}(\mathrm{sp}(true, \texttt{assume } x < 0), x := -x) \to x \geq 0.$$

For demonstration, we can mix these two approaches to check

$$\mathrm{sp}(true, \texttt{assume } x < 0) \to \mathrm{wp}(x \geq 0, x := -x)$$

The antecedent turns into $true \wedge x < 0$, while the conclusion turns into $-x \geq 0$. Simplifying the first in FOL and the second in the underlying theory gives, $x < 0 \to x < 0$, which is valid. Hence, we succeeding in verifying the @post condition of `abs`. Notice that we verified the implication of the postcondition from the precondition for all paths in `abs`.

3 Basic Paths & Inductive Assertion

When we relabel the @pre-condition with π_s and the @post-condition with π_e, the approach in the last section formalizes as

$$p \in P[s, e] : \mathrm{sp}(\pi_s, p) \to \pi_e.$$

The function obeys it specification if for all paths from s to e the strongest postcondition of the precondition implies the function's post condition.

We fulfilled this proof obligation exactly this way when proving the postcondition of `abs` and it was quite easy, as there are only finitely many paths. In the general case involving loops, however, the number of paths through a flow graph is – from a static point of view –

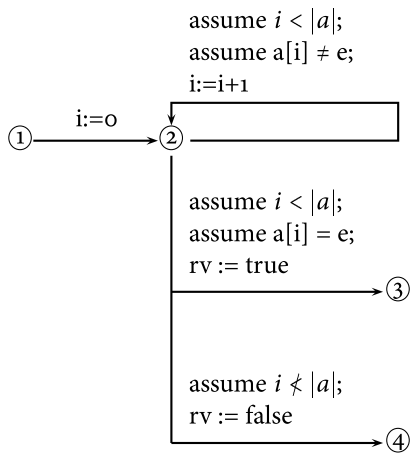


Figure 3: The flow graph of LinearSearch

infinite, $|P[s,e]| = \infty$. The flow graph[3] of LinearSearch given in Figure 3 demonstrates this problem, as its simple loop over the array creates an infinite number of paths. The check-all-paths approach fails here.

To cut down this infinite amount of work to a finite amount, the verifying compiler limits its scope of operation to non-looping subparts. It only checks paths of the flow graph that start and end with a loop or with the functions start/end. In this examples these paths are ①-②, ②-②, ②-③, ②-④. They are basic in the sense that all other paths through the flow graph can be put together from these paths as building blocks. Hence, these paths are called *basic paths*. This work load reduction does not come for free. For this technique to work we need to provide assertion at every flow graph cut.

Before applying this technique we proof it correct. Therefore we introduce a bit of helpful notation: $P[s,e]$ denotes the set of all path from the node s to e. $\langle s,\dots,e\rangle$ is an element of this set. $\langle s,\dots,\cancel{n},\dots,e\rangle$ is a path that does not pass by the node n – that is, the path does not contain n as node except maybe as start and end point. $\langle s,(\dots,n)^i,\dots,e\rangle$ is a path that passes by the node n i-times.

Lemma 1. *If*
$$a \to b$$
then
$$sp(a,p) \to sp(b,p)$$

[3]Notice this flow graph does not contain any flow control constructs. Its paths solely consists of `assume` and `assign` statements.

Proof. By the definitions of sp. □

Lemma 2. *If sp is a valid strongest postcondition transformer and*
$$sp(x,\langle a,\dots,b\rangle) \to y$$
$$sp(y,\langle b,\dots,c\rangle) \to z$$
then
$$sp(x,\langle a,\dots,b,\dots,c\rangle) \to z$$
for the combined path.

Proof. From $sp(x,\langle a,\dots,b,\dots,c\rangle)$ deduce $sp(sp(a,\langle a,\dots,b\rangle),\langle b,\dots,c\rangle)$ by the recursive definition of sp. Weaken the inner sp expression to y by applying Lemma 1. z follows directly from the resulting expression with the second assumption.

□

Lemma 3. *If*
$$\forall p \in P[s,n] : (p = \langle s,\dots,\cancel{n},\dots,n\rangle \wedge sp(\pi_s,p)) \to \pi_n) \quad \wedge$$
$$\forall p \in P[n,n] : (p = \langle n,\dots,\cancel{n},\dots,n\rangle \wedge sp(\pi_n,p)) \to \pi_n)$$
then
$$\forall p \in P[s,n] : sp(\pi_s,p) \to \pi_n$$

Proof. Inductive proof on the number of occurrences of n in p. Assume $p = \langle s,(\dots,n)^i\rangle$. $i = 1$ by the first assumption. We use the induction hypothesis
$$sp(\pi_s,\langle s,(\dots,n)^i\rangle) \to \pi_n,$$
and the second assumption as the assumptions for Lemma 2 to conclude
$$sp(\pi_s,\langle s,(\dots,n)^{i+1}\rangle) \to \pi_n.$$
□

Theorem 4. *If*
$$\forall p \in P[s,e] : (p = \langle s,\dots,\cancel{n},\dots,e\rangle \wedge sp(\pi_s,p)) \to \pi_e) \quad \wedge$$
$$\forall p \in P[s,n] : (p = \langle s,\dots,\cancel{n},\dots,n\rangle \wedge sp(\pi_s,p)) \to \pi_n) \quad \wedge$$
$$\forall p \in P[n,n] : (p = \langle n,\dots,\cancel{n},\dots,n\rangle \wedge sp(\pi_n,p)) \to \pi_n) \quad \wedge$$
$$\forall p \in P[n,e] : (p = \langle n,\dots,\cancel{n},\dots,e\rangle \wedge sp(\pi_n,p)) \to \pi_e)$$
then
$$\forall p \in P[s,e] : sp(\pi_s,p) \to \pi_e$$

Proof. Assume $p = \langle s, (\dots, n)^i, \dots, e\rangle$. The first assumption justifies the case $i = 0$. For $i \geq 1$, we use the second and third assumption to apply Lemma 3 to conclude $sp(\pi_s, \langle s(\dots, n)^i\rangle \to \pi_n$. This in conjunction with the last assumption, serves as assumption to Lemma 2 to give $sp(\pi_s, p) \to \pi_e$.

□

The validity of the basic path verification method is a corollary of this theorem. Any node in the flow graph of a Pi-program is the start of a loop, so when we apply the theorem for every node, we end up with proof obligations equivalent to our verification conditions around basic paths. Note that the last theorem is not only valid for flow graphs constructed from Pi-programs but for arbitrary shaped programs. Also graph cutting can start at inner loops as the proof obligation for paths from n to n does not only involve the loop body but also cycles of any outer loop even around involving s and. e.

Thanks to Theorem 4 we can replace an infinite amount of paths involving ②, such as ①-②-..., ①-②-②-..., ①-②-②-②-... by the proof obligations of Theorem 4. Fulfilling the first requirement, $\langle s, \dots, \cancel{n}, \dots, e\rangle$, is easy, because this path set is empty as there are no paths that do not pass over n. The second theorem requirement demands a check for ①-②, the third for a check for ②-②, and the last requirement two checks: ②-③ and ②-④. These paths are exactly the set of basic paths for this flow graph. But where does the assertion π_n come from to generate the verification conditions? Theorem 4 itself does not give any hints.

We can use the requirement for the path ①-② as a guide and just use the strongest postcondition of ① as π_n, so the required implication holds. $sp(true, i := 0)$ asserts $i = 0$ at ② when arriving from ①. However, this choice as π_n is problematic, as the next proof requirement for the path ②-② is not fulfilled. The VC

$\{i := 0\}$`assume ...; assume ...; i:=i+1`$\{i := 0\}$

is invalid. We need a stronger predicate π_n to start with, so that we can validate the VC from ② to ②, and further proof the postcondition at ③ and ④.

As we demonstrate later, a suitable assertion can not be found automatically and the programmer has to supply an assertion by annotating the loop. If the VC around the loop is valid and further supports the postconditions, we call the predicate *inductive*. Notice, that many trivial predicates are inductive when we look only at the loop body. For instance, the @post-condition would not be derivable from the assertion *true* which is clearly inductive but only with respect to the loop body. A predicate that is sufficiently strong is

$$\forall j : 0 \leq j < i : a[j] \neq e$$

Looking at all the proof obligations reveals, that this assertion is indeed inductive and supports the post condition. We only review the reasoning for ②-②.

$\forall j : 0 \leq j < i : a[j] \neq e$

```
assume i < |a|;
assume a[i] != e;
i := i+1;
```

$\forall j : 0 \leq j < i : a[j] \neq e$

Moving the precondition forward with sp over the two assumption statements gives:

$\forall j : 0 \leq j < i : a[j] \neq e \wedge i < |a| \wedge a[i] \neq e$

```
i:=i+1;
```

$\forall j : 0 \leq j < i : a[j] \neq e$

The final step gives:

$\forall j : 0 \leq j < i : a[j] \neq e$

```
assume i < |a|;
```

$a[i] \neq e \to \forall j : 0 \leq j < (i+1) : a[j] \neq e$

We can drop the last assumption as it does not support the conclusion. We arrive at

$$\forall j : 0 \leq j < i : a[j] \neq e \to \\ (a[i] \neq e \to \forall j : 0 \leq j < (i+1) : a[j] \neq e)$$

which is valid. We can verify that the VC for ②-③ and ②-④ holds by using a similar reasoning with wp and sp.

Once we have a potentially inductive assertion, checking it can be fully automatic, when we stick to decidable fragments of FOL.

4 Intuition for preconditions and postconditions

Diving into the non-trivial forward propagation algorithm in the next section is easier when we have an intuitive idea how predicates relate to each other with terms like weaker/stronger and how to derive weaken/strengthen predicates. Figure 4 gives an oversimplified two dimensional representation of predicates.The objects on this pane are states that fulfills predicates when they are encircled by them. Predicates effectively model sets and when we draw them in set-diagram style

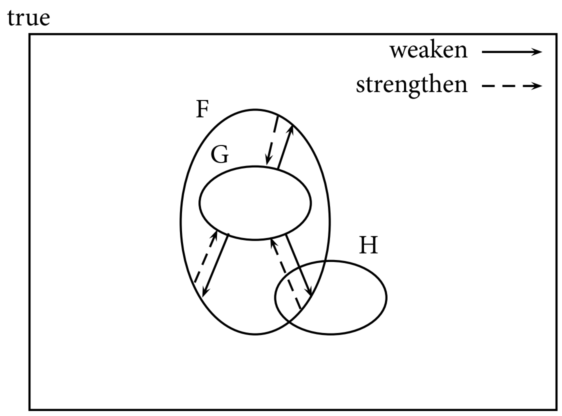


Figure 4: The space of state predicates

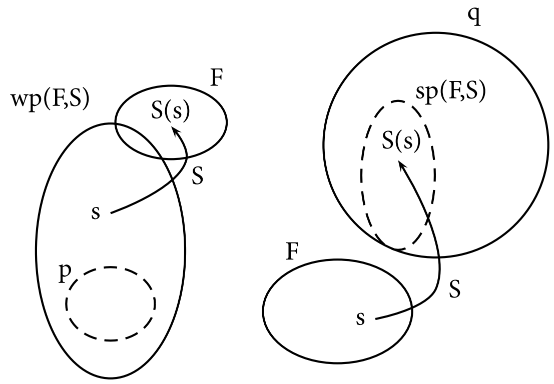


Figure 5: Weakest precondition with stronger predicate p and strongest postcondition with weaker predicate q

like in Figure 4, the relationships between predicates are the same as between sets. The subset relations is captured by the implication, union by logical *or*, and intersection by logical *and*. The most trivial predicate *true* is equal to the universe of discourse as it admits all states to its set. The smallest element is the *false* predicate which is equal to the empty set. As the empty set is a subset of every other set, *false* trivially implies every other predicate. Validating the implication $G \rightarrow F$ checks whether G is stronger then F (or whether F is weaker than G respectively).

Figure 5 gives an idea how weak/strong pre/postconditions relate to each other. Inspecting the diagram for the strongest postcondition on the right side, let us deduce that if we have a valid postcondition then we are able to proof any other postcondition to be valid if we can show that it is weaker than our original. We do that by implication checking – or if we want to stick to the set-theoretic description by superset checking. By choosing the strongest postcondition (the smallest in cardinality), we effectively describe all valid postconditions under implication.

Likewise, a valid precondition always implies the validity of stronger preconditions. By choosing the weakest precondition, we effectively describe all valid preconditions with the inverse implication, or subset checking.

5 Forward Propagation

Focusing only on basic paths succeeds in solving the verification problem with the help of human ingenuity to provide the respective inductive assertions at loops. But is there a heuristic that at least in some cases come up with the strongest @post-condition of a function on its own? A possible naive strategy for this would be to push down the strongest postcondition from the entry nodes until we reached all exit nodes, and then join the conditions at the exit nodes by $\vee$ to deduce the strongest postcondition. For flow graphs without loops or with loops that can be unrolled this approach works quite well. But how about the general case?

Before we investigate on how to construct a solution, we have to gain some insight into how to verify a proposed solution. Assume a proposed solution is given as a function π that maps nodes into assertions. We call this function assertion hypothesis. If it contains a node assertion that is not implied by the strongest postcondition that arrives from one of the node's predecessors, then we successfully falsified the hypothesis. If we can not find such a node then we have a valid solution.

Figure 6 models these two situations. We assume p is the assertion at the node currently inspected, F is the assertion of one of the predecessors and $\mathrm{sp}(F, S)$ the strongest postcondition computed from the statements on the path coming from the predecessors. On its left side the current assertion p is a proper one, as the strongest postcondition arriving from F is inside of the set of states described by p. Hence, there is nothing contradicting the validity of p at the node inspected. On the right hand side, we see a successful falsification. The strongest postcondition is not a subset of the states described by p.

In general, if we have an initial solution, a verification procedure and an improvement procedure, we can easily construct an algorithm that iteratively computes

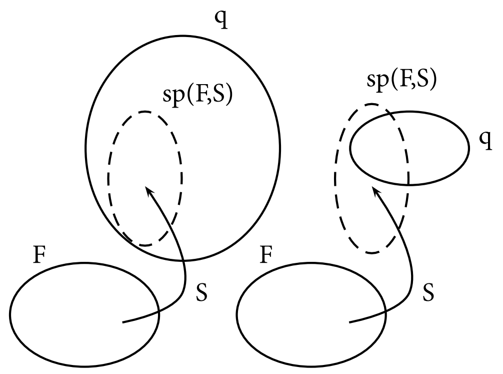


Figure 6: *Left:* All states implied by the strongest postconditions are covered by q. *Right:* Not all states are covered. q must be weakened.

```
@pre true
@post ??
bool LimLinSearch(int[] a,int e) {
  for (int i:=0; i<3; i:=i+1) {
    if(a[i]==e) return true;
  }
  return false;
}
```

Figure 7: Limited Linear Search

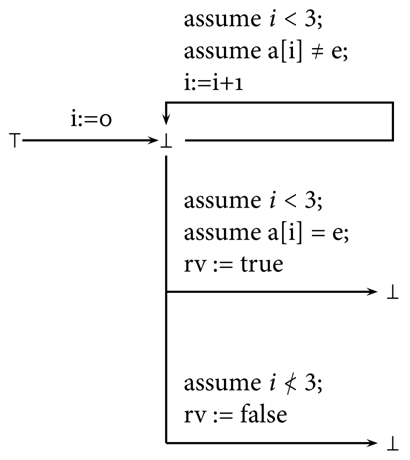


Figure 8: Initial state for forward propagation in LimLinSearch

solutions.[4] To apply this technique, we identify the trivial initial hypothesis, that maps all nodes to *false*, except the start node, which is mapped to @pre. If we have *false* as node assertion, this can be read as the assertion that this node is unreachable for the control flow, namely its predecessors and successor paths contain dead code.

When the verification procedure falsified our hypothesis, we can retry with an improved hypothesis. Assume the verification procedure rejected the hypothesis because for some node b

$$sp(\pi_a, \langle a, b\rangle) \nrightarrow \pi_b.$$

We can improve π by replacing π_b with $\pi_b \vee sp(\pi_a, \langle a, b\rangle)$. This makes the assertion weaker and allows for more states, effectively covering the states implied by predecessor's strongest postconditions. Hence the implication that failed to hold, now holds.

Let us see this approach in action with a simple example. We create a limited version of linear search in Figure 7 that is only able to search the first three elements of a list.[5] The flow graph of `LimLinSearch` given in Figure 8 is identical to `LinearSearch`'s flow graph. It also contains the troublesome loop, but now bounded by 3 instead $|a|$. In Figure 8, we replaced the nodes with the initial hypothetical assertions.

Our first target for falsification is the assertion of node ②. Coming from ①, we compute $sp(\top, i := 0)$ which is $i = 0$. As $\perp$ does not imply $i = 0$ we have to adjoin $i = 0$ to $\perp$ by $\vee$ to improve the hypothesis. $i = 0 \vee \perp$ simplifies to just $i = 0$ which becomes the new assertion for ②.

As we changed the assertion at ②, new potential falsification targets pop up, namely the successors of ②, nodes ③, ④ and ② itself. As we have to conduct a few iterations of pushing results from ② to ②, we wait with processing ③ and ④ until we are finished with ②.

Computing the strongest postcondition for $i := 0$ under the statements

```
assume i<3; assume a[i]≠e; i:=i+1
```

gives

$$i := 1 \wedge (i-1) < 3 \wedge a[(i-1)] \neq e, \qquad (*)$$

which simplifies to $i := 1 \wedge a[0] \neq e$. The current assertion at ②, $i := 0$ does not imply (*). Hence, we have to weaken the assertion at ② and turn it into

$$i := 0 \vee (i := 1 \wedge a[0] \neq e).$$

[4]Termination is not guaranteed

[5]We intentionally omit the requirement that the list contains at least three elements.

After another propagation from ② to ②, we get

$$(i := 1 \wedge a[0] \neq e) \vee \underbrace{(i := 2 \wedge a[0] \neq e \wedge a[1] \neq e)}_{P},$$

of which only the first part is supported by the assertion at ②, but P is not. Hence, we weaken the assertion at ② and adjoin P to the assertion of ② by a logical *or*. Another round of propagation at ② gives:

$$\begin{aligned}&(i := 1 \wedge a[0] \neq e) \vee (i := 2 \wedge a[0] \neq e \wedge a[1] \neq e) \vee \\ &\quad \underbrace{(i := 3 \wedge a[0] \neq e \wedge a[1] \neq e \wedge a[2] \neq e)}_{P} \quad (^{**})\end{aligned}$$

Again the assertion of ② supports only the first parts of this postcondition, but not P. Hence, it is added with logical *or* as before.

The next iteration from ② to ② is different as we reached the statically known bound for the loop variable i. With previous iterations, we have seen that computing the strongest postcondition always rediscovers parts that are already at ② and a new $\vee$-connected fragment. At our current iteration, the rediscovered part is exactly $(^{**})$ while the new yet unsimplified fragment is

$$i = 4 \wedge (i-1) < 3 \wedge a[i-1] \neq e$$

But in contrast to previous iterations, this formula is unsatisfiable, hence it simplifies to false. So, we have to check whether

$$\big((^{**}) \vee \bot\big) \Rightarrow \big(i = 0 \vee (^{**})\big).$$

As this implications is valid, we reached a fixed point under this process at ② with the final assertion at ② being

$$\begin{pmatrix} & (i = 0) \\ \vee & (i = 1 \wedge a[0] \neq e) \\ \vee & (i = 2 \wedge a[0] \neq e \wedge a[1] \neq e) \\ \vee & (i = 3 \wedge a[0] \neq e \wedge a[1] \neq e \wedge a[2] \neq e) \end{pmatrix} \quad (\dagger)$$

We can continue with checking ③ and ④. Propagating $(\dagger)$ over `assume` i `<` `3; assume` $a[i]$ $\neq$ e; `rv:=true` gives,

$$(\dagger) \wedge \; rv = true \wedge a[i] = e \wedge i < 3$$

We can simplify this by noticing that i can only be 0, 1, 2. Expanding all cases of i in $a[i] = e$ and dropping the case $i = 3$, results in

$$\begin{aligned}rv \wedge (a[0] = e \vee (a[0] \neq e \wedge a[1] = e) \vee \\ (a[0] \neq e \wedge a[1] \neq e \wedge a[2] = e))\end{aligned}$$

We proceed likewise for the postcondition of ④. The strongest postcondition over `assume` i $\nless$ `3;` `rv:=false` gives

$$(\dagger) \wedge i \nless 3 \wedge \neg rv$$

From $(\dagger)$ only the last fragment is valid in conjunction with $i \geq 3$, hence we get

$$\neg rv \wedge i := 3 \wedge a[0] \neq e \wedge a[1] \neq e \wedge a[2] \neq e$$

$\vee$-connecting the conditions at ③ and ④ and dropping the internal variable i gives us the strongest function's postcondition.

6 The algorithm and its Termination Criteria

What we have done informally in the last example, is captured in the forward propagation algorithm given in ML syntax in Figure 9.

For a node n, `p1` n ss ah ws pushes down the respective strongest postconditions to all successors of n contained in the list ss. When finished, `p1` returns a tuple (ah', ws') of the new assertion hypothesis ah' and a new workset ws'. ws' is the original workset ws plus all nodes that received new assertions. In Line 5, `p1` checks whether a successor node needs new assertion or whether the existing one q suffices. If q is implied by the newly propagated strongest postcondition p, there is no further work for this successor (Line 6). In Figure 6, this is the case on the left side. If q is not implied by p, then the strongest postcondition covers states that are not in q (right side in Figure 6). q must be weakened to support these states by adjoining the strongest postcondition with logical *or*. We change the assertion q at the successor node with the help of function `mod` which modifies ah such that (`mod` *ah node ass*) *node* = *ass*. Additionally, the successor is put into the workset, so that the newer and weaker assertion can be pushed down further to the successors of this successor.

As long as nodes need inspection – that is the maintained workset ws is non-empty – `p*` carries out pushes for these nodes by calling `p1`. The initial call is

`p* ((mod` $a_\bot$ e pre`), (`e`::nil))`

```
p1 nil w ah ws' = (ah, ws')
 | (s::ss) w ah ws' =
    let val p = (sp_A (ah w) (path w s))
        val q  = (ah s)
        in if (⇒_A p q) then
            p1 ss ah w ws'
        else
            p1 ss (mod ah s (⊔_A p q))
               w (s::ws')

p* (ah, nil) = ah
 | (ah, w :: ws) =
    p* (p1 w (succ w) ah ws)
```

Figure 9: Forward Propagation

where $a_\perp$ is a function so that $\forall n : a_\perp\ n = \perp_\mathcal{A}$, e is the entry node and *pre* the function's @pre-condition. If this expression ever finishes evaluating, p* returns a valid assertion map that contains the strongest postconditions at exit nodes with inductive assertions at loop nodes.

The forward propagation acts upon a static analysis algebra. Such an algebra $\mathcal{A}$ needs a carrier set $\mathbb{D}$ with functions defined over the following signatures:

$$\begin{array}{ll} \text{sp} :: \mathbb{D} \to \mathbb{S}^* \to \mathbb{D} & \perp :: \mathbb{D} \\ \Rightarrow :: \mathbb{D} \to \mathbb{D} \to \mathbb{B} & \top :: \mathbb{D} \\ \sqcup :: \mathbb{D} \to \mathbb{D} \to \mathbb{D} & \end{array}$$

where $\mathbb{S}$ is the set of valid statements, and $\mathbb{B}$ the set of boolean values. sp is the strongest postcondition transformer. Note that we index the transformer with F when we refer to the strongest postcondition for the FOL domain, as we defined it in Section 2. $\Rightarrow$ implements domain specific implication checking. $\sqcup$ joins elements so that, $F \Rightarrow (G \sqcup H)$ must imply $F \Rightarrow G$ or $F \Rightarrow H$. $\perp$ is the smallest (strongest) element of $\mathbb{D}$ used to initialise in the initial assertion hypothesis, while $\top$ is the largest (weakest) element of the domain.

Using FOL as static analysis algebra naturally maps $(\mathbb{D}, \text{sp}, \Rightarrow, \sqcup, \perp)$ to $(F, \text{sp}_F, \to, \vee, \textit{false})$ where F is the set of valid FOL formulas. But FOL is a problematic choice as static analysis algebra. In general, when used with forward propagation it does not guarantee the termination of $p*$. Remember from LimLinSearch that each iteration around ② caused a new fragment of knowledge about states at ② to become available. Usually, it had the form $i = n \wedge (i-1) < |a| \wedge a[i-1] \neq e$. We had to adjoin this new knowledge to the assertion of ② until we encountered a boundary for i. If this boundary is statically unknown – as it is with the expression $i < |a|$ in LinearSearch – we are never able to conclude that $i < |a|$ can not hold as we were able with $(i-1) < 3$ when we had arrived at $i = 4$ in LimLinSearch. Hence, the loop of knowledge propagation around ② never stops for LinearSearch. Notice that with FOL, forward propagate is nothing else than symbolic interpretation.

Scrutinizing the assertion generated in non-terminated cases reveals that these node assertions are infinitely ascending chains in $\Rightarrow$. Whether there exists infinite ascending chains depends on the domain and the join operator. In the next section, we examine two different algebras, namely Interval Analysis and Karr's Analysis. Both terminate when used with forward propagation, however when compared to FOL, they are not as precise. Therefore, the assertion derived from both analysis might not be the strongest @post-condition that a human might be able to derive.

7 Applied Analysis: Interval Analysis

In contrast to the preciseness of FOL, Interval Analysis (IA) is a very rough description of program states. IA describes a program state by associating each variable with an interval $[a, b]$, where $a, b \in \mathbb{Q}^+$ and $\mathbb{Q}^+ = \mathbb{Q} \cup \{-\infty, \infty\}$. These state functions serve as the domain $\mathbb{D}_I$,

$$\mathbb{D} = \{x, y, \dots\} \to [\mathbb{Q}^+, \mathbb{Q}^+].$$

Such a function maps variables x,y into intervals. We use the regular substitution notation to express modifications of this function. If there is no proper knowledge about a variable, its interval is $[-\infty, \infty]$. $\top_I$ is a function that maps all variables to the interval $[-\infty, \infty]$. If any variable maps to $[\infty, -\infty]$ the state can be simplified to $\perp_I$, the smallest element of the domain. Its meaning is similar to FOL's *false*, namely it labels unreachable code.

For intervals, we define two projection functions that are typographically written as over- and underlined expressions:

$$\underline{[a, b]} := a \qquad \overline{[a, b]} := b$$

Next we define an interval arithmetic evaluation function

$$[\![e]\!] : \mathbb{A} \to \mathbb{D}_I \to [\mathbb{Q}^+, \mathbb{Q}^+]$$

that maps arithmetic expressions in $\mathbb{A}$ into intervals with the help of the state function. The arithmetic expressions of π are covered as follows:

Scalar values c: $[\![c]\!]s = [c, c]$

Variables x: $[\![x]\!]s = s\ x$.

Linear functions : Unary function f in $\mathbb{Q}$ can be lifted cheaply to interval arithmetic

$$[\![f(x)]\!]s = [\min(f(a), f(b)), \max(f(a), f(b))]$$

where $[\![x]\!]s = [a, b]$. This works likewise for a linear binary function $\odot$:

$$[\![x \odot y]\!]s = [\min(a_1 \odot b_1, a_1 \odot b_2, a_2 \odot b_1, a_2 \odot b_2), \max(a_1 \odot b_1, a_1 \odot b_2, a_2 \odot b_1, a_2 \odot b_2)]$$

where $[a_1, b_1] = [\![x]\!]s$ and $[a_2, b_2] = [\![y]\!]s$. Likewise this works with n-array functions with the respective number of argument permutations.

Non-linear functions like x^2 evaluate to $[-\infty, \infty]$.

Mappings from existing FOL to IA and vice verse is important as this is π's annotation language.

$$\bigwedge_{v \in \mathrm{Dom}(F)} \underline{s(v)} < v \wedge v < \overline{s(v)}$$

is the respective FOL representation of a IA state function. Mapping FOL formulas into IA only works for those fragments that are $\wedge$-connected and have the form $x \leq y$ or $x = y$. To save a bit of definition work, we assume that

- $x = y$ is expanded into $x \leq y \wedge y \leq x$,
- $\wedge$-connected FOL expressions are simplified to a series of `assume` statements, and
- variables occurs as the only literals on its side of an inequality, e.g. $5 < 10x$ gets rewritten to $\frac{1}{2} < x$.

With these tricks it is sufficient to define only,

$$\mathrm{sp}_I(s, \texttt{assume } c \leq x) := s[x/[\max(c, \underline{s\ x}), \overline{s\ x}]]$$

$$\mathrm{sp}_I(s, \texttt{assume } x \leq c) := s[x/[\underline{s\ x}, \min(c, \overline{s\ x})]]$$

Assignment is the evaluation of the arithmetic expression e in the original program state s followed by replacing the assigned variable in the program state s with the evaluation result:

$$\mathrm{sp}_I(s, \texttt{x:=}e) := s[x/[\![e]\!]s]$$

A state s implies a weaker state s' if every variable in s' is mapped to a bigger interval than the respective mapping in s.

$$s \Rightarrow s' \iff \forall v \in \mathrm{Dom}(s') : s\ v \subseteq s'\ v$$

As the subset relation is decidable by inspecting the intervals bounds, implication checking in IA is decidable.

Finally, we define the join operator in the IA algebra:

$$(s \sqcup s')x = [\min(\underline{s\ x}, \underline{s'\ x}), \max(\overline{s\ x}, \overline{s'\ x})]$$

A final problem with interval arithmetic remains, namely that the no-ascending chain property is not guaranteed. The following simple loop example demonstrates this problem:

```
assume i = 0;
assume n ≥ 0;
while (i<n) {
  i:=i+1
}
```

The initial interval for i is $[0, 0]$. The strongest postcondition of the loop body is $[1, 1]$ for i after one iteration. As the initial interval $[0, 0]$ does not imply $[1, 1]$, we have to join these two intervals: $[0, 1]$. Another loop iteration gives $[1, 2]$ as new interval, which is again not implied by $[0, 1]$. Hence, we adjoin it to get to $[0, 2]$ and so on. As the loop is bound by a statically unknown variable n, this process never stops.

Therefore, we redefine the join operator a bit to overestimate the interval of its variables non-deterministically,

$$(s \sqcup_I s')x = \begin{cases} (s \sqcup s')x & \text{if widen() = false} \\ [l, u] & \text{if widen() = true} \end{cases}$$

with

$$l = \begin{cases} -\infty & \underline{s'\ x} < \underline{s\ x} \\ \underline{s\ x} & \text{otherwise} \end{cases} \qquad u = \begin{cases} \infty & \overline{s'\ x} > \overline{sx} \\ \overline{s\ x} & \text{otherwise} \end{cases}$$

With the help of this sometimes overestimating join operate, interval analysis infers correctly that i has no statically known upper bound. Although for loops with statically known but large boundaries as $i < 1000$, IA might overestimate the range of i.

8 Applied Analysis: Karr's Analysis

Karr's Analysis captures linear relations among variables of the form

$$c_0 + c_1 x_1 + \cdots + c_n x_n = 0 \qquad (4)$$

for $c_i \in \mathbb{Q}$ and the program variables x_i. This analysis rests on the concept of affine spaces. There are two equivalent representations of affine space. Either as a series of constraints in the form (4) or as a set of vectors $V = \{v_1, \dots, v_k\}$ that generates the space with

$$\mathit{affine}(V) = \left\{ \sum_i \lambda_i \overline{v}_i \right\} \qquad \text{where } \sum_i \lambda_i = 1$$

The vector set is easier to manipulate so we choose vector sets to represent program states in this analysis. By solving the linear equation system

$$\begin{pmatrix} \overline{V} \\ \overline{1} \end{pmatrix} \overline{\lambda} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{pmatrix}$$

we can convert the vector representation back into a constraint representation.

For simplicity, we disregard the information contained within assume statements, and define the strongest postcondition sp_K only over assignments. Furthermore, we have to restrict our attention to affine assignment which have the form

$$x_k := c_0 + c_1 x_1 + \cdots + c_n x_n.$$

To change the program state captured as the set of vectors, we apply the following transformation function

$$f(\overline{x}) = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & \vdots & & \\ c_1 & c_2 & \cdots & c_n \\ & \vdots & & \\ & & & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ c_0 \\ \vdots \\ 0 \end{pmatrix} \qquad (5)$$

to all elements of the set. If the assignment contains a non-affine expression, we conduct two transformations $x_k := 0$ and $x_k := 1$ and join the resulting vectors sets. The result is that the variable x_k is unbound in the affine space.

The $\sqcup$ operator is defined as the union of the set-union of both vector sets. The smallest element $\bot_K$ is the empty set, , while the largest element $\top_K$ is the set of unit vectors. For vectors with three elements, this is:

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\} \qquad (6)$$

Implication checking, $F \Rightarrow G$ is decidable by checking whether all vectors in F are in affine(G). This is equivalent to finding vectors $\overline{\lambda}$ such

$$\forall v \in F. \exists \overline{\lambda}. \begin{pmatrix} G \\ \overline{1}^T \end{pmatrix} \overline{\lambda} = \begin{pmatrix} \overline{v} \\ 1 \end{pmatrix}$$

Satisfying this equation for $\overline{\lambda}$ is efficiently decidable by using Gaussian elimination. If all $\overline{v} \in F$ are in affine(G) then, $F \Rightarrow G$.

Consider the following example

```
i=0; j=0; k=0;
while (?) {
  k:=k+1;
  if (??)
    i:=i+1;
  else
    j:=j+1
}
```

As we will show, Karr's Analysis can discover the inductive invariant $i + j = k$ at the loop's body. There are three basic paths here: the initial block of assignments, the loop's body with the conditional's then-branch and the loop's body with the false-branch.

According to (5), the combined transformations from the initial assignment is

$$\tau_1 \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Applying this transformation to all elements of the unit vector (6), reduces the set of vector to just $\overline{0}$:

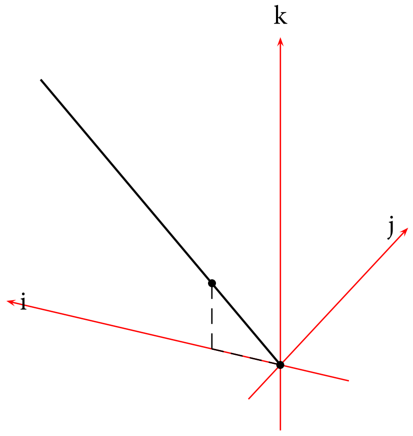
$$\{\overline{0}\} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

The transformations for the other two basic paths, `k:=k+1; i:=i+1` and `k:=k+1;j:=j+1`, have the following to transformations:

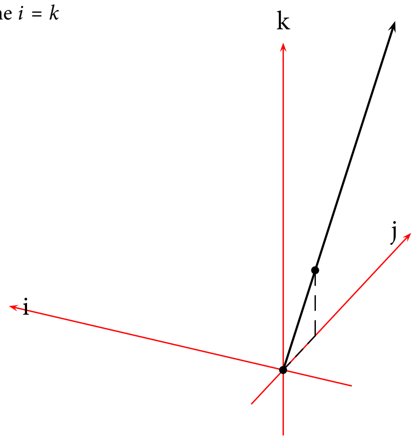
$$\tau_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\tau_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

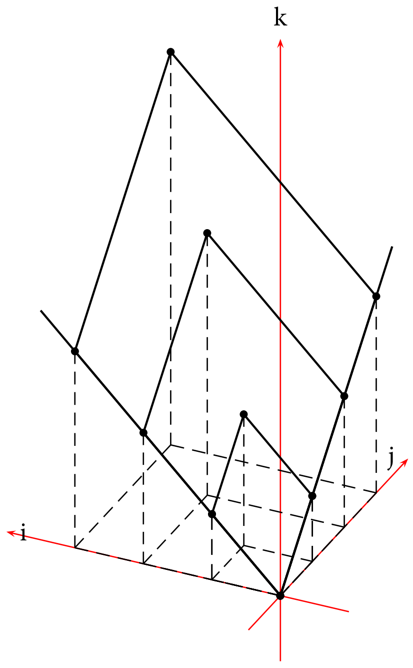
Although the forward propagation algorithm propagates to one branch and then to the other, we treat both



(a) The vector set $\{(0,0,0)^T,(1,0,1)^T\}$ projects the line $i = k$



(b) The vector set $\{(0,0,0)^T,(0,1,1)^T\}$ projects the line $j = k$



(c) The vector set $\{(0,0,0)^T,(1,0,1)^T,(0,1,1)^T\}$ projects the plane $i + j = k$

Figure 10: Affine spaces induces by vector combinations

in parallel, as this make the symmetry in the arguments more obvious. Applying τ_2 and τ_3 transformations to $\overline{0}$ gives:

$$\tau_2\overline{0} = \begin{pmatrix}1\\0\\1\end{pmatrix} \qquad \tau_3\overline{0} = \begin{pmatrix}0\\1\\1\end{pmatrix}$$

As both are not implied by $\overline{0}$ we add both results to the asserted vector set at the loop. Figure 10 displays the affine space induced by combining $\overline{0}$ with these results one at a time. The then-branch gives $\{\overline{0},\tau_2\overline{0}\}$ and if converted back to constraints gives $i = k$ as restriction. This restriction is clearly visible in (a). The false-branch gives $\{\overline{0},\tau_3\overline{0}\}$ which results in the restriction $j = k$ as seen in (b). Combining these two restrictions induces a pane in the affine space described by $i + j = k$, as seen in (c).

We observe that

$$\tau_2\tau_3\overline{0} = \tau_3\tau_2\overline{0}$$

so the reiterating both new results with the respective other transformation gives the same result $(1 \quad 1 \quad 2)^T$ however this results is redundant,

$$(-1)\begin{pmatrix}0\\0\\0\end{pmatrix} + 1\begin{pmatrix}1\\0\\1\end{pmatrix} + 1\begin{pmatrix}0\\1\\1\end{pmatrix} = \begin{pmatrix}1\\1\\2\end{pmatrix}$$

Reiterating further with τ_2 and τ_3 gives the redundant vectors

$$\begin{pmatrix}2\\0\\2\end{pmatrix} = 2\begin{pmatrix}1\\0\\1\end{pmatrix} + (-1)\begin{pmatrix}0\\0\\0\end{pmatrix}$$
$$\begin{pmatrix}0\\2\\2\end{pmatrix} = 2\begin{pmatrix}0\\1\\1\end{pmatrix} + (-1)\begin{pmatrix}0\\0\\0\end{pmatrix}$$

Hence we have found an inductive annotation at (2):

$$\left\{\begin{pmatrix}0\\0\\0\end{pmatrix},\begin{pmatrix}1\\0\\1\end{pmatrix},\begin{pmatrix}0\\1\\1\end{pmatrix}\right\},$$

which translates into the restriction $i + j = k$.

Bibliography

[BM07] Bradley and Manna. The Calculus of Computation. 2007.