Spin — Superposition with Structural Induction

Andreas Halkjær From

June 2019

1 Introduction

In this report, I describe my work to develop a Viper mode for the GAPT project based on techniques from Superposition with Structural Induction by Simon Cruanes [1].

The goal is to solve files from the Tons of Inductive Problems (TIP) set. Spin currently solves 142 of these when given a 2 minute time limit. GAPT already includes a parser for the TIP format and Jannik Vierling has developed support for compiling function definitions into conditional rewrite rules to enable my work. Where Cruanes' implements definitions with if-then-else using a new definition with an extra argument matching on the condition, GAPT implements them as conditional rules instead. That is, a definition is only allowed to unfold if the conditions all reduce to true.

Spin analyses these rewrite rules to determine which arguments to perform induction on when generating axioms. I describe the injection of axioms in Section 2 and the argument analysis in Section 3. I take a look at the generated axioms and motivate some heuristics in Section 4.

Contrary to Cruanes' Zipperposition, Spin does not use the rewrite system during saturation which relies purely on the Escargot superposition prover. Another difference between Spin and Zipperposition is how we test conjectures before trying to prove them. Zipperposition relies on saturation with a limited number of steps whereas Spin inserts concrete sample terms in place of the variables to induct over and normalises the resulting formulas using the rewrite system. I describe this machinery in Section 5.

I have evaluated the performance of Spin on the TIP library and compared it to the other Viper modes. It performs strictly better than the analytic modes. I describe these results in Section 6.

I will mention future work when relevant in the corresponding section. A notable task would be to make use of AVATAR for lemma introduction as outlined in section 5 of Cruanes' paper [1].

As an overview, Spin implements the following features and heuristics:

- Injection of inductive axioms based on selected clauses during saturation and not just the initial goal.

- Analysis of primary, accumulator and passive arguments of definitions to guide the generation of axioms.
- Induction over multiple primary arguments simultaneously when they occur together under the same defined symbol.
- Generalisation of subgoals in two ways:
  - Induction only over primary occurences, leaving passive occurences alone.
  - Induction over whole subterms, not just inductive skolem constants, that occur in primary positions.
- Universal quantification of accumulator arguments in the induction target.

2 Injecting Axioms

I start by describing how the Spin mode adds inductions to the proving process.

This happens in two places, first Spin analyses the initial goal and adds any axioms deemed appropriate to the left-hand side of the sequent before passing it over to the superposition prover. The reason for this is that the goal may contain implications causing the goal to be split across multiple clauses when converted to CNF. Thus, the dependency is lost since Spin only deals with one clause at a time.

Next, every time a clause is picked by the saturation process, Spin analyses the clause and possibly generates one or more axioms for proving the negation of the clause.[1] These axioms are stored in an ordered queue and periodically added to the working set of clauses. This happens either when reaching saturation or when the saturation loop has run a number of times, starting at 16, with this cutoff increasing every time we have added five axioms to account for them.

In case a proof is found, the added axioms are then cut out of this proof.

3 Argument Analysis

The purpose of argument analysis is to determine which arguments of a function should be the target of induction. For a motivating example, consider the following axiomatisation of addition over natural numbers built from constants $Z/0$ and $S/1$:

$$Z + y = y \qquad S(x) + y = S(x + y)$$

[1]Axioms in the initial sequent are also analysed, but should not produce any axioms as the negation would fail tests. An optimisation would be to note their origin and not analyse them at all.

Say we wish to prove the following property of addition:

$$x + S(y) = S(x + y)$$

This requires an induction over x, because $+$ is defined over its first argument. If we attempt an induction over y, both the base case and inductive step are stuck immediately.

The analysis is performed first over the individual rules of each definition and then put together for the entire definition. I will use the prefix *locally* for analysis within individual rules. This split into local and global analysis is my own invention. Cruanes uses phrases like "every occurrence" where it is unclear whether there needs to be at least one occurrence or not [1, Definition 1].

I first describe the analysis as implemented in Spin and then outline the differences from Cruanes' definition which is very short.

3.1 Local rule analysis

In general, a rule of a definition of f/n has the following shape:

$$s_1, \dots, s_k \to f(t_1, \dots, t_n) = u$$

where the s_i are conditions.

The argument positions of f are denoted by their index, so that term t_i appears at position i. Note that u may contain both recursive calls and calls to other functions. The conditions may contain calls to other functions.[2]

There are four possibilities for a position: locally undetermined, locally passive, locally an accumulator and locally primary.

Locally undetermined A position i is locally undetermined if the rule does not contain a recursive call. I use this category to simplify the global analysis.

Locally passive A position i is *locally passive* if every occurence of f in u has t_i as argument. That is, it is passed unchanged in the recursive call. Furthermore, any variable which is a subterm of t_i may only occur in passive positions of calls to other functions in $s_1, \dots, s_k$ and u. Position 2 of the axiomatisation of $+$ given above is locally passive in both rules.

In the following rule, $==$ is assumed primary in both positions so position 1 is forced by the condition to be primary, even though e is passed unchanged in the recursive call.

$$x == e \to count(e, x :: xs) = S(count(e, xs))$$

[2]The latter is not covered by Cruanes.

Locally an accumulator A position i is *locally an accumulator* if it is not locally passive, t_i is a variable and t_i does not appear in primary position to another call.

Position 2 is locally an accumulator in the following rule for reversing a list using an accumulator:

$$rev(x :: xs, acc) = rev(xs, x :: acc)$$

Locally primary A position is locally primary in all other cases. E.g. position 1 in $S(x) + y = S(x + y)$ because it changes in the recursive call.

3.2 Global rule analysis

I often omit the "globally" prefix.

An argument position for a collection of rules defining f is:

Globally passive if it is locally passive in at least one rule and all either locally undetermined or locally passive in all other rules.

Globally an accumulator if it is not globally passive and locally an accumulator across all rules.

Globally primary otherwise.

Note that here I have collapsed the undetermined positions into primary ones. This means that definitions which are only abbreviations, e.g. $rev(xs) = rev\text{-}with\text{-}acc(xs, [])$, have their positions treated as primary.

It also means that the single argument of a definition $id(x) = x$ is treated as primary, which may not be what you want, so there is room for refinement here. Notably, that definition should probably be viewed as undetermined as well and then the analysis should depend on the context in which it appears: $id(x) = x$ vs. $id(x + y) = y + x$.

3.3 Occurences

A primary occurence is a position nested only under primary positions or uninterpreted symbols. As we have seen, terms in primary occurrences are the ones to induct on. An accumulator occurrence is not primary but nested under primary and accumulator positions. A passive occurrence is nested under some passive position.

3.4 Mutual recursion

The analysis above depends on having already analysed any function which is called by the function under analysis. Spin computes a topological order on-the-fly to do this (the analysis only happens once in the beginning so performance does not matter). Some functions are mutually recursive and in this case, each of them is analysed by treating every argument position of the other as primary. Future work includes possibly inlining definitions for a few terms to break the cycle and do the regular analysis.

4 Generated Axioms

By default, Spin inducts on inductive skolem constants that appear in primary positions.

So given the clause:

$$s_1 + S(s_2) \neq S(s_1 + s_2)$$

where s_1, s_2 are skolem constants, Spin analyses that s_1 has two primary occurrences and s_2 two passive ones. Therefore, the conjecture generated by negating the formula and generalising primary occurrences becomes:

$$\forall x.\ x + S(s_2) = S(x + s_2)$$

Spin tests a fixed number of instances of this formula and since the tests all pass, injects the following axiom into the saturation loop:

$$\begin{aligned}
&Z + S(s_2) = S(Z + s_2) \wedge \\
&(\forall x.\ x + S(s_2) = S(x + s_2) \rightarrow S(x) + S(s_2) = S(S(x) + s_2)) \rightarrow \\
&\forall x.\ x + S(s_2) = S(x + s_2)
\end{aligned}$$

4.1 Heuristics

This section gives motivating examples for the heuristics implemented in Spin. Only the last one, accumulator arguments, is not given by Cruanes.

4.1.1 Primary occurrences under same defined symbol

Consider the clause:

$$l \neq \mathit{take}(n, l){+}{+}\mathit{drop}(n, l)$$

where $\mathit{take}(n, l)$ returns the first n elements of l, $\mathit{drop}(n, l)$ drops the first n elements from l and ++ appends two lists.

The arguments n and l have two primary occurrences each. For each of them, they occur directly under the same defined symbol, either *take* or *drop*. This suggests that we should do an induction over both arguments simultaneously. The intuition is that the definition does not unfold otherwise. That is, we should try to prove:

$$\forall l. \forall n.\ l = \mathit{take}(n, l){+}{+}\mathit{drop}(n, l)$$

I use the sequential axiom machinery in GAPT for this which generates two induction axioms. The first one by induction over the outer quantifier with the inner one preserved, and the next one over the inner one.

This corresponds to first inducting over l for arbitrary n and then in each case inducting over n.

4.1.2 Generalising subgoals

Each of the generalised subgoals implies the original subgoal, so superposition will automatically derive the latter from the former.

Passive occurences Consider the clause:

$$\underline{s_1} + (s_1 + s_1) \neq (\underline{s_1} + s_1) + s_1$$

The constant s_1 has two primary occurrences, marked by an underline, so that suggests we should induct on it. But if we do so directly, generalising all occurrences we get the goal:

$$\forall x.\ x + (x + x) = (x + x) + x \tag{*}$$

The base case is provable, but in the inductive step, the constructors in passive positions will prevent application of the induction hypothesis (equivalent to (*)) since the case reduces to:

$$S(x + S(x + S(x))) = S(x + S(x)) + S(x)$$

Therefore, if the term has at least two primary and one passive occurrence, Spin will conjecture a goal where only the primary occurrences are generalised:

$$\forall x.\ x + (s_1 + s_1) = (x + s_1) + s_1$$

This passes tests and is more general than the other one, so Spin will inject the axiom generated by induction on x into the saturation step instead. Crucially, this one does not have the problem of constructor symbols blocking application of the induction hypothesis.

Whole subterms Consider the clause:

$$\underline{rev(s_1)}{+}{+}(s_2{+}{+}s_3) \neq (\underline{rev(s_1)}{+}{+}s_2){+}{+}s_3$$

The two primary occurrences are underlined. If we only induct on the constant s_1, the definition of rev will unnecessarily start to unfold. Instead, because $rev(s_1)$ has at least two primary occcurrences, we check if the following generalised conjecture passes tests:

$$\forall xs.\ xs{+}{+}(s_2{+}{+}s_3) = (xs{+}{+}s_2){+}{+}s_3$$

It does and follows straight-forwardly by induction over xs.

4.2 Accumulator arguments

A heuristic not mentioned by Cruanes is to universally quantify over accumulator arguments before applying the induction principle.

Consider the following tail-recursive axiomatisation of addition:

$$Z +' y = y \quad S(x) +' y = x +' S(y)$$

Here, the first argument position is primary and the second an accumulator. Consider the following clause, obtained by negating the goal that the two axiomatisations behave equivalently:

$$s_1 +' s_2 \neq s_1 + s_2$$

Without the considered heuristic, we attempt to refute this by proving the following by induction on x:

$$\forall x.\ x +' s_2 = x + s_2$$

But in the step case we have:

$$\forall x.\ x +' s_2 = x + s_2 \to S(x) +' s_2 = S(x) + s_2$$

where the consequent reduces to:

$$x +' S(s_2) = S(x + s_2)$$

Due to the constructor that s_2 occurs under, we cannot apply the induction hypothesis. Instead, we prove *this goal* by induction on x:

$$\forall x.\ \forall y.\ x +' s_2 = x + s_2$$

Then the step case becomes:

$$\forall x.\ (\forall y.\ x +' s_2 = x + s_2) \to (\forall y.\ S(x) +' y = S(x) + y)$$

and we can instantiate the hypothesis at the appropriate term.

5 Testing Conjectures

Spin tests any conjecture it creates an induction axiom for, in order to avoid wasting time attempting to prove false things. This testing is performed by inserting concrete sample terms for the variables inducted on and normalising the resulting formula. This raises a few issues, that I describe in this section.

5.1 Unblocking

The normalisation depends on the reduction rules generated for the definitions in the given problem. Sometimes however, these are stuck even though all possible inputs would reduce equally.

As an example, consider axiomatised equality for the natural numbers:

$$Z == Z \rightsquigarrow \top \quad S(x) == S(y) \rightsquigarrow x == y$$

$$Z == S(x) \rightsquigarrow \bot \quad S(x) == Z \rightsquigarrow \bot$$

During unfolding of other definitions we may end up with a term like $s_1 == s_1$ where s_1 is a constant. The equality does not reduce to true even though it would for all concrete inputs.

When encountering these terms, Spin substitutes the constant with a concrete representative of each constructor. In this case Z and $S(Z)$. If all such substitutions normalise to true, the term is accepted. There is a risk of accepting a term that should not have been accepted, a false positive, but this is only a performance issue as the formula still has to be proved by superposition.[3]

5.2 Data type rules

Reduction rules for function definitions is not enough, Spin also needs disjointness and injectivity for data type constructorss as well as reflexive equality for each type. So these rules are generated at the beginning for the given problem.

5.3 Unfolding quantifiers

Consider the following formula with an existential quantifier:

$$elem(s_1, l) \to \exists k.\ !!(l, k) = s_1$$

where $!!(l, k)$ denotes the k'th element in list l.

When substituting concrete lists for the variable l, the existential quantifier remains. To successfully test formulae like this, Spin *unfolds* the quantifier into a concrete number of tests. In particular the same number of tests as the number of terms l is sampled by. The resulting formula becomes:

$$elem(s_1, l) \to !!(l, Z) = s_1 \lor !!(l, S(Z)) = s_1 \lor !!(l, S(S(Z))) = s_1 \lor \ldots$$

Universal quantifiers are unfolded to conjunctions.

Clearly this is a conservative technique that may provide false negatives in case we do not sample the correct witness for the existential or fail to sample the failing example for the universal. Thus, completeness may suffer.

[3]In reality, it is more of an issue since generalised subgoals are preferred over originals, so if the generalised subgoal falsely passes tests, it will be the target of induction instead of the original subgoal.

5.4 Process

Testing a formula proceeds as follows: Quantifiers are unfolded and the result is normalised by the given reduction rules as well as by simple elimination of $\top$ and $\bot$. If the formula reduces to $\top$ then it is accepted and if it reduces to $\bot$ then it is rejected. Otherwise, the formula is passed to a SAT solver and accepted if it returns valid. If not and the formula still contains function terms, we try the unblocking technique and recursively test the resulting formulae. If this still fails there are two cases: Either the normalised formula contains function terms or not. If it does not, the formula is rejected but if it does, it is only conditionally rejected.

Conditionally rejected formulae are accepted in case the formula contains higher-order functions, as Spin cannot construct samples for functions.

6 Experiments

The timeout for all experiments are 120 seconds to find a proof and return it.

6.1 Mode comparison

Figure 1 shows which TIP problems are solved by which Viper modes. The mode spin_nogen is Spin without generalisation and spin_notest is Spin where candidates for proof by induction are not tested. The size of each point illustrates the total time spent searching for a proof. As evident, Spin solves every file solved by the other Spin modes and the analytic modes. Table 1 shows the number of files solved by each mode.

Only “isaplanner/prop_59.smt2” is not solved by Spin but solved by another mode, namely the treegrammar mode. The goal of “prop_59” is:

$$\forall xs.\ \forall ys.\ ((ys : list) = nil \to last(\mathord{++}(xs, ys)) = last(xs))$$

The reason Spin does not solve it, is that it requires a nested induction because of the definition of *last*. But at that point the needed hypothesis $ys = nil$ has been split into another clause and “lost.” I think Cruanes addresses this as follows [1, sec. 6.1, bottom]:

> If an induction variable j is a sub-case of some other constant, there exists induction hypotheses (in other clauses) that might be needed for nested induction. In this case we also try the inductive goal where j is *not* replaced by a fresh variable and run both proof attempts simultaneously.

However, it is not entirely clear to me what sub-case means here and I only looked into it late, so implementing it is left for future work.

ana. ind.	ana. seq.	spin	spin no gen.	spin no test.	treegr.
78	115	142	133	94	24

Table 1: Number of TIP files solved by each mode

6.2 Axiom generation

While the analytic modes only add inductions to the initial sequent, Spin also generates them during saturation. Figure 2 shows the time spent on generating axioms out of the total time spent. The average ratio of axiom generation to total search time was 0.27 with the minimum being 0.012 and maximum 0.60.

6.3 Inductions in cleaned proof

Figure 3 shows the number of inductions in the proof obtained from Spin after this proof has been converted to an expansion proof and back. The total number of inductions in the proof is given in black, the number of atomic inductive axioms used in the cleaned proof is in red and the number with quantifiers in blue. Notably only "tip2015/list_nat_elem.smt2" uses quantified axioms.

The induction depth, that is the number of inductive inferences applied above each other in the cleaned proofs, is always 1. I think this has to do with the roundtrip to an expansion proof which turns nested inductions into lemmas that are then cut away[4].

On average, Spin proofs contain 0.69 inductions per inductive axiom added during the proof search. In other words, only every third axiom is added in vain.

Figure 4 compares the number of inductions in proofs found by Spin and the analytic sequential mode. On average, Spin proofs contain 1.87 inductions where the proofs found using the sequential axioms contain 1.64.

6.4 Cost of generalisation

Unfortunately, while generalisation causes nine additional files to be solved it sometimes slows down Spin significantly. The details are given in Figure 5.

Spin spends longer on axiom generation because more clauses than just those with inductive skolem constants are considered. For Spin without generalisation, the ratio of axiom generation to total time is on average 0.13, where for normal Spin it was 0.27. But more time is also spent during saturation. The mean time across the files solved by both modes is 13.3 seconds for Spin with generalisation and 10.9 seconds for Spin without.

[4]Assuming each induction corresponds to one axiom, which mostly seems to be the case but should be investigated.

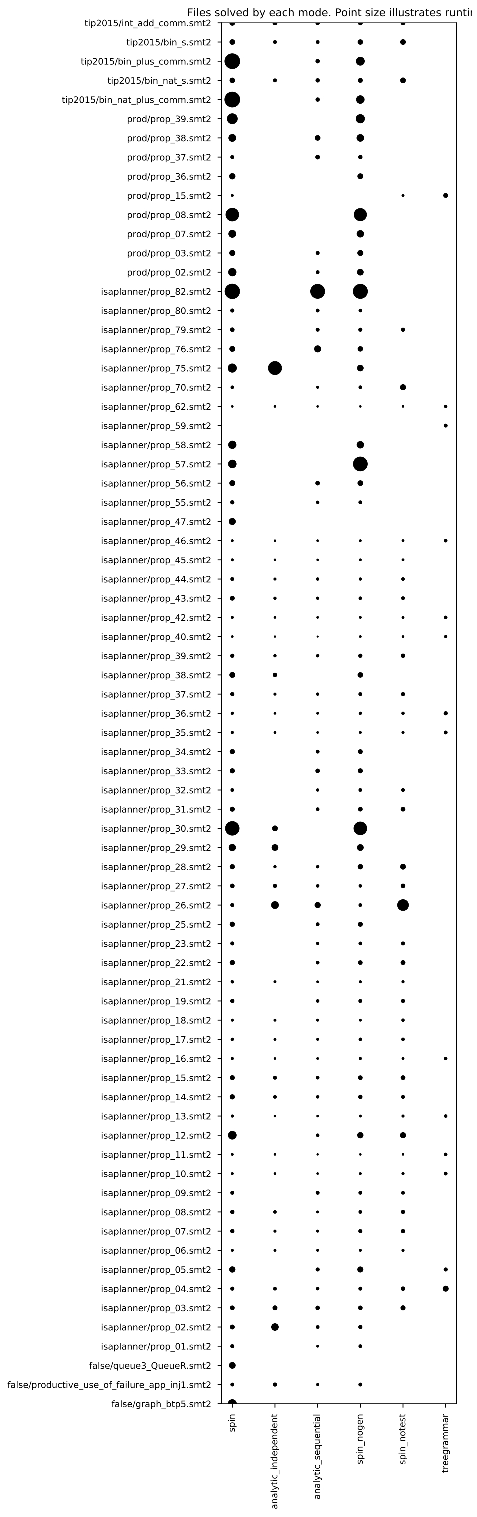
Cruanes mentions the need to heuristically limit generalisation for this reason. Spin does not currently implement such limiting.

6.5 Benefit of testing

Figure 6 shows the clear benefit of testing, in that a lot fewer axioms are injected while still finding a proof. In fact, Spin solves 142 problems with testing but only 94 without due to either falsely accepting generalised subgoals or slowing down the search process too much to fit within the timeout.

References

[1] Simon Cruanes. ‘Superposition with Structural Induction’. In: *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings.* 2017, pp. 172–188. DOI: `10.1007/978-3-319-66167-4\_10`.



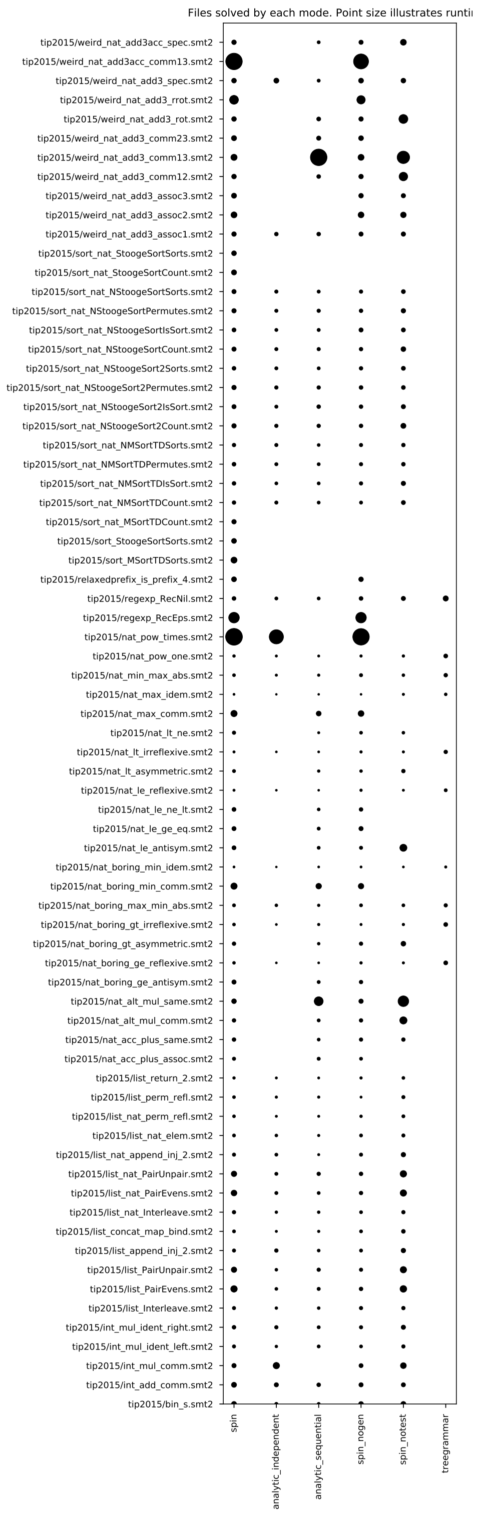


Figure 1: Scatter plot of solved files by the various modes.

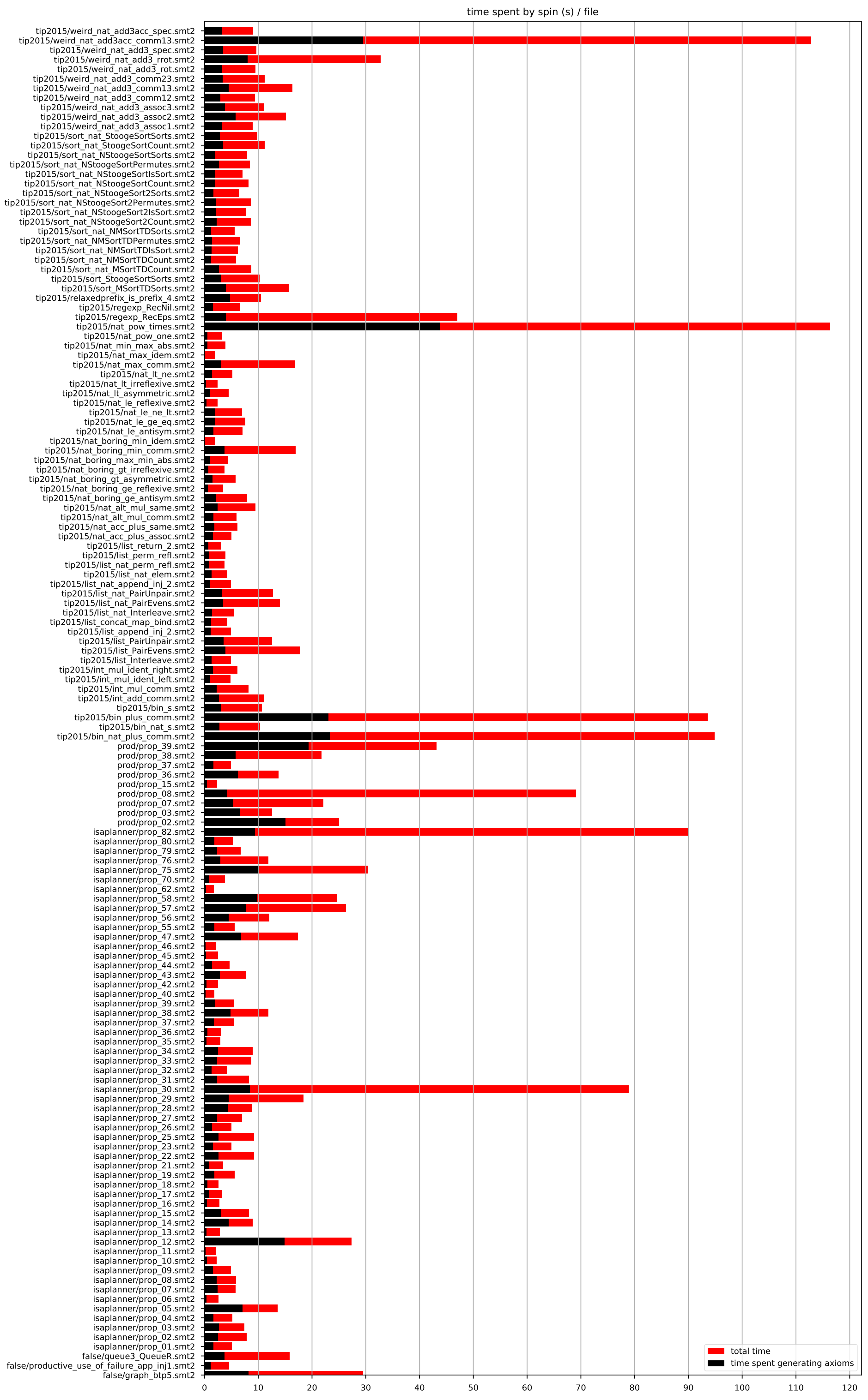


Figure 2: Total time in red; proportion spent on axiom generation in black.

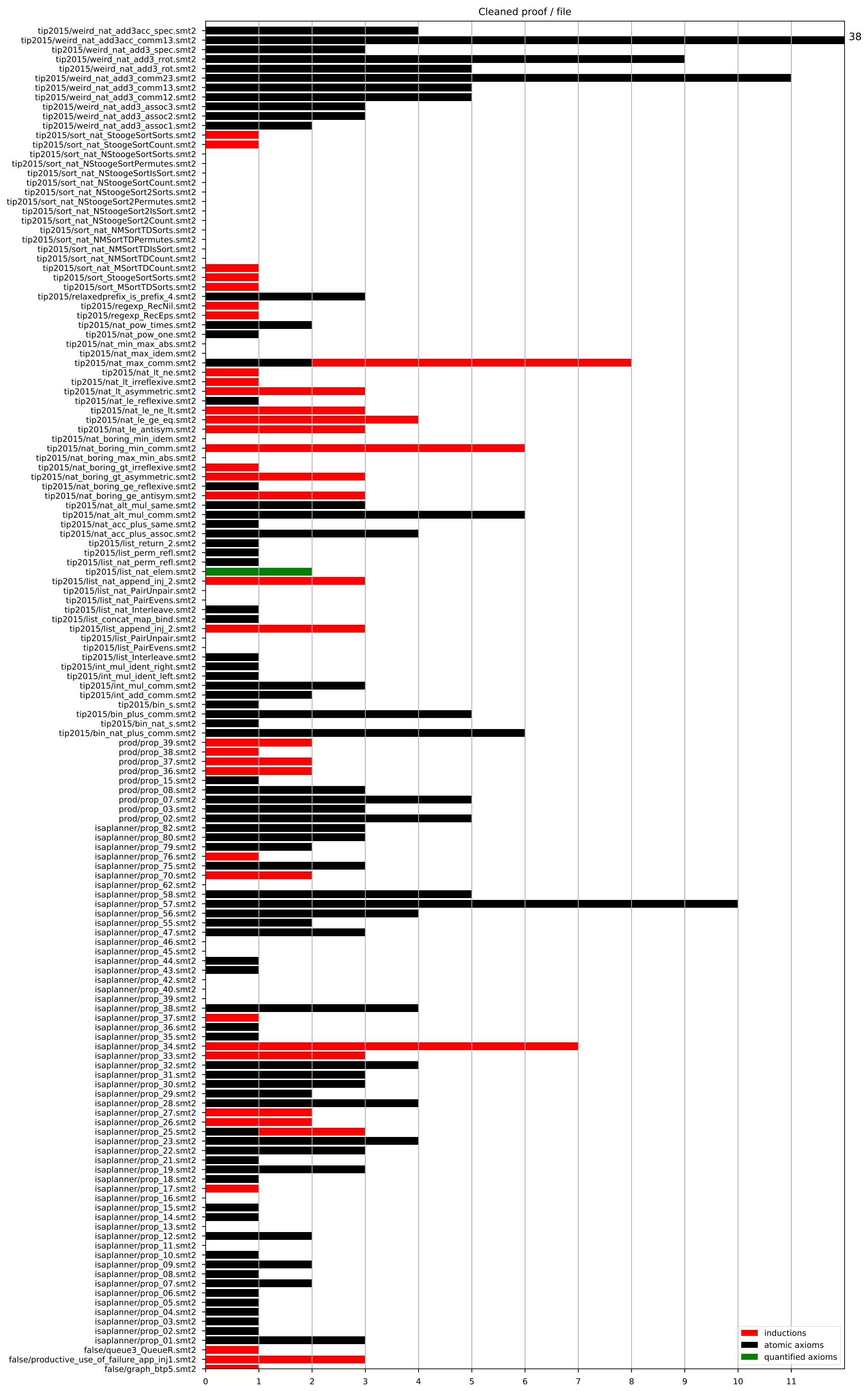


Figure 3: Total number of inductions in red, proportion of atomic axioms in black and proportion of quantified axioms in green.

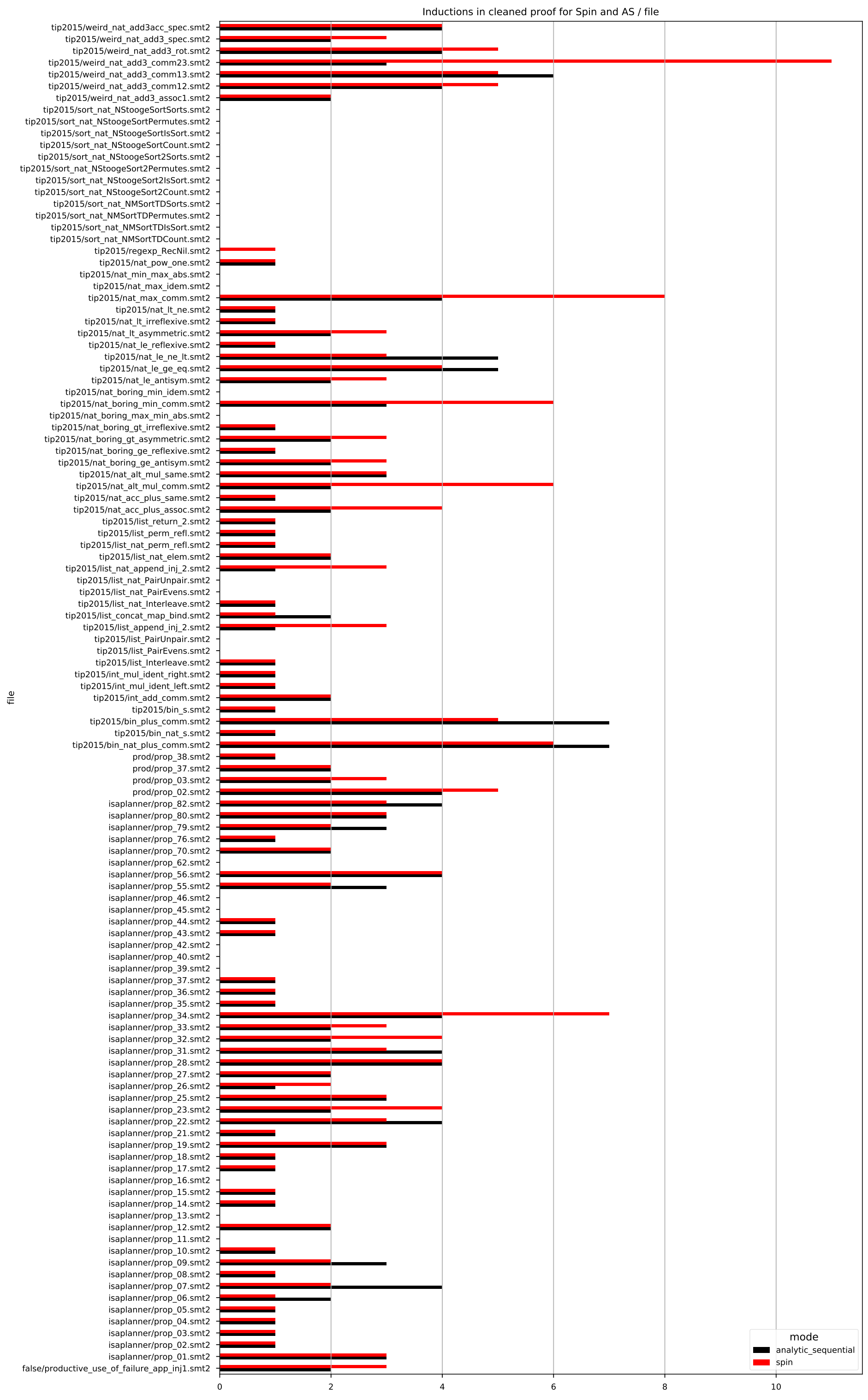


Figure 4: Inductions in cleaned proofs by Spin and analytic sequential mode.

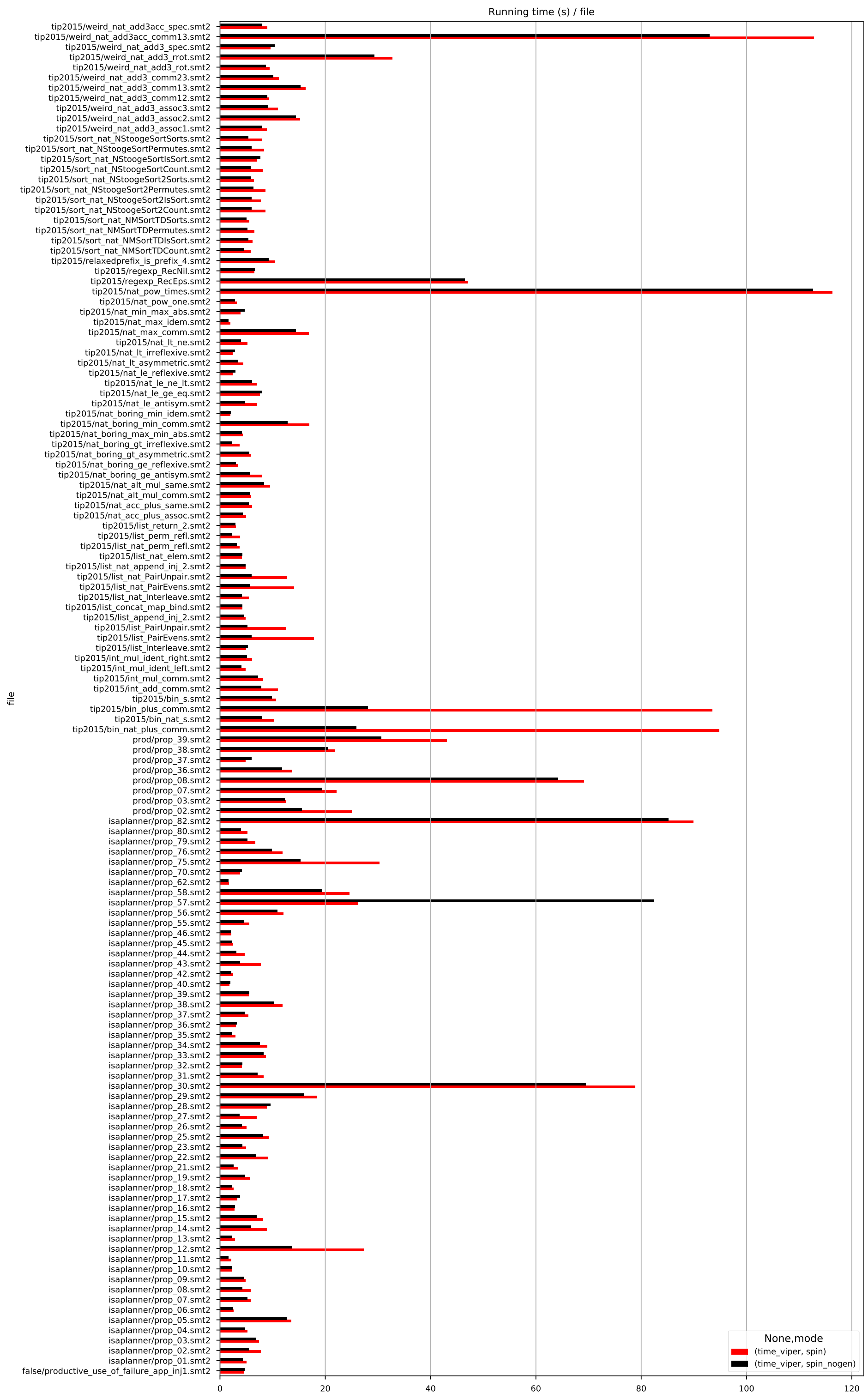


Figure 5: Comparison of runtimes for Spin with and without generalisation.

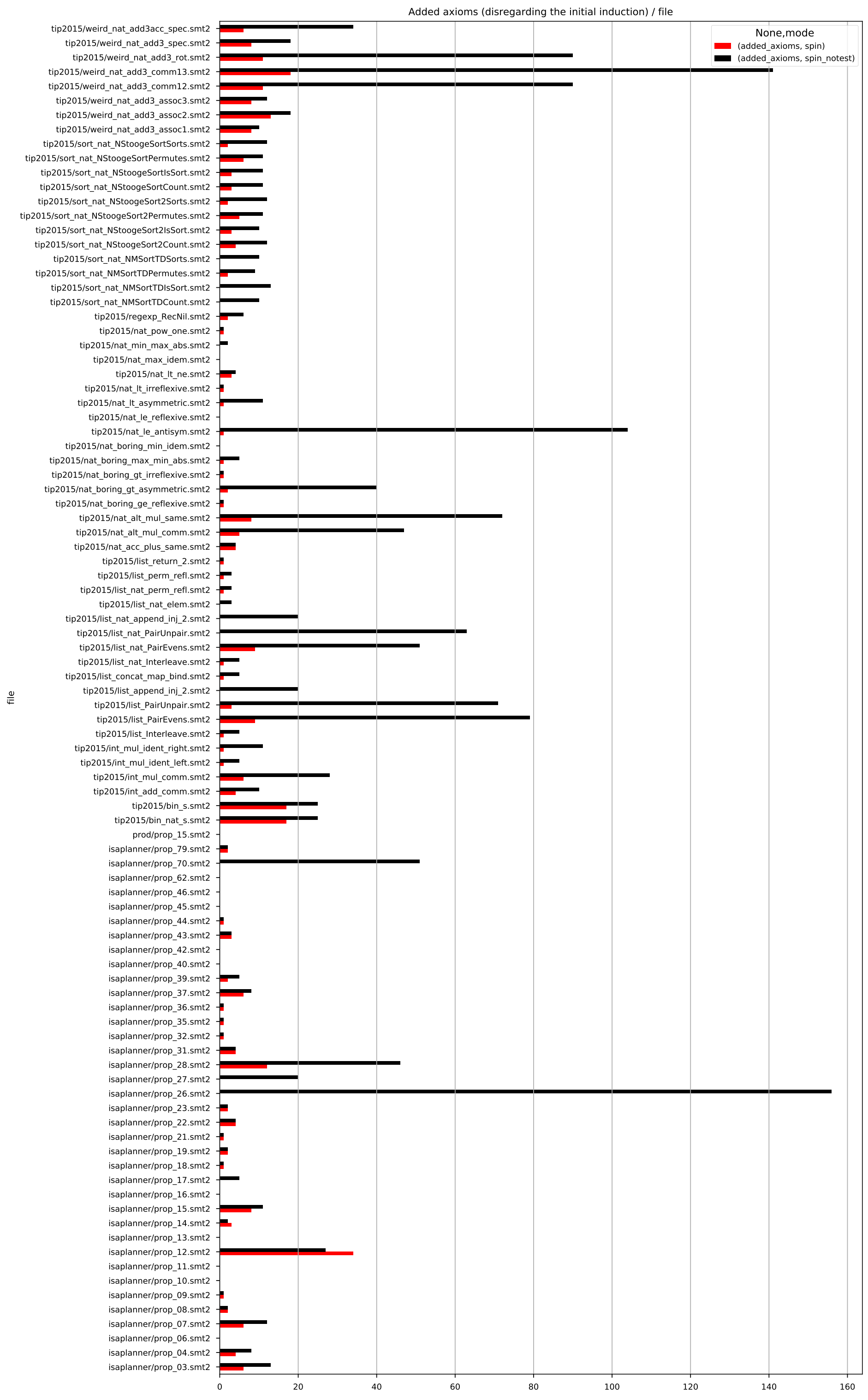


Figure 6: Axioms injected by Spin with and without testing.