Quantifiers meet their match(ing loop): new techniques and tools for dealing with unpredictable performance in Dafny

Clément Pit-Claudel (MIT CSAIL), K. Rustan M. Leino (MSR)

July 11, 2016

What is Dafny?

- Dafny is a verification-aware programming language.
- Like many other tools, Dafny is based on Boogie and Z3.
- It runs on most platforms, and has advanced editor support in Visual Studio and (now!) in Emacs.

Dafny hands-on: finding the largest value in a sequence (solution)

What problem are we trying to solve?

- Dafny is very snappy on small programs
- Larger programs tend to suffer from butterfly effects:
  - Verification performance is chaotic (unstable and unpredictable)
  - Insignificant source changes cause verification failures

```
var x := y;
assert …;
```

✓ Verifies :)

```
var x := y + 0;
assert …;
```

✗ Fails to verify ?!

This work focuses on this issue in the context of Dafny, but we expect it to generalize to other verifiers.

What causes instability?

- Translation
  - Similar Dafny programs can look very different at the Z3 level
- Undecidable/Semi-decidable domains:
  - Non-linear arithmetic
  - **First-order logic (quantifier instantiations) ⟵ Our focus**
    - Matching loops
    - Costly instantiations

How can we make things better?

- Educate users: Implement IDE facilities to encourage users to write Z3-friendly theorems
- Improve the general experience: This one problem won't be as noticeable if everything else goes smoothly
- Improve the debugging experience: Help advanced users diagnose instabilities with better debugging tools
- Address the underlying problem:
  - **Choose better triggers for quantifiers**
  - **Try to prevent matching loops**
  - Improve translation

Improving predictability while maintaining soundness requires relinquishing some amount of completeness.

What did we implement?

A bit of all!

- We improved Dafny by adding trigger-related facilities:
  - matching pattern (trigger) generation
  - matching loop elimination
  - quantifier splitting

and also:

- We created Emacs modes for Dafny, Boogie, and a bit of Z3
- We extended the Z3 axiom profiler to add produce new interactive graphical visualizations (ask me for a demo!)
- We added tooltips and warnings in cases where we don't have good automatic fixes for triggering issues

The Dafny pipeline

0. Parse
1. Type-check
2. Transform the AST ⟵ this project happens here
3. Translate to Boogie
4. Translate to Z3
5. Verify

How does Z3 handle quantifiers?

- Z3 relies on triggers (matching patterns) to instantiate quantifiers. Every time Z3 comes across a new term, it instantiates all quantifiers whose triggers match the new terms. For example:

`IsHuman(Socrates)`
$\forall$ `h {IsMortal(h)}` $\cdot$ `IsHuman(h)` $\Longrightarrow$ `IsMortal(h)`
`Goal: IsMortal(Socrates)`

- Bad trigger choices cause verification failures, matching loops, and costly instantiations.
- Z3 knows how to pick good triggers for clean formulas.

Why isn't this enough?

Z3 produces excessively liberal triggers on Dafny programs.

- Dafny produces large formulas with many parasitic terms, due to its internal encoding.
  - Dafny: `s[x]`
  - Boogie: `$Unbox(read($Heap, s#0, IndexField(x#1)))`
  - Z3: `(U_2_int ($Unbox intType (MapType1Select $Heap1 |s#00| …)))`
- Ad hoc fixes improve the situation, but only to some extent.
- Debugging and understanding trigger choices is hard (triggers are Z3 terms, not Dafny terms!).

How do we generate good triggers?

0. Walk the AST below a quantifier. Annotate each term as
   - A trigger head, if it can act as a trigger:
     `f(x)` `old(h(x, y))` `x in multiset{1,2}`
   - A trigger killer, if it prevents parent nodes from being heads:
     `x+1` `¬y` `x in multiset{1,2}`
1. Collect all trigger heads
2. Compute the power set to generate all possible multi-triggers
3. Reject invalid multi-triggers (not mentioning all variables)
4. Filter for efficiency

Trigger generation example

Quantifier: $\forall\ x \cdot P(x) \wedge (Q(x) \implies P(x+1))$

Subexpressions: `x P(x) Q(x) 1 x+1 P(x+1) (Q(x)⟹P(x+1)) …`

Killers: `x+1 P(x+1) (Q(x)⟹P(x+1)) …`

Heads: `P(x) Q(x)`

- The resulting triggers are made of Dafny terms that appear in the body of the quantifier.
- Heuristics are used to reduce the set of triggers under consideration (issues arise with quantifiers over many variables).

What do we gain?

- Triggers now come from actual Dafny terms: we can show them to the user directly
- Parasitic terms are not chosen as triggers anymore: less costly instantiations
- We can show warnings when we can't find good triggers
- And we can start looking for matching loops!

What are matching loops?

- Matching loops occur when a instantiating a quantifier (or a set of quantifiers) produce terms that directly or indirectly cause it to be instantiated again, repeatedly:

$\forall\ x\ \{f(x)\} \cdot f(x) \leq \mathbf{f(f(x))}$

$f(0) \rightsquigarrow f(f(0)) \rightsquigarrow f(f(f(0))) \rightsquigarrow f(f(f(f(0)))) \rightsquigarrow f(f(f(f(f(0))))) \rightsquigarrow \ldots$

$\forall\ x\ \{P(x)\} \cdot P(x) \wedge (Q(x) \Longrightarrow \mathbf{P(x+1)})$

$P(x) \rightsquigarrow P(x+1) \rightsquigarrow P(x+2) \rightsquigarrow P(x+3) \rightsquigarrow P(x+4) \rightsquigarrow \ldots$

Detecting and suppressing matching loops

0. For every candidate trigger, compute the set of matching terms in the body of the quantifier.
1. For each matching term, decide whether it looks like a loop:
   - `{f(x)}` ⋪ `f(x)` ? Safe
   - `{f(x)}` ⋪ `f(x+1)` ? Loops
   - `{f(x)}` ⋪ `f(f(x))` ? Loops
   - `{f(f(x))}` ⋪ `f(x)` ? Safe
   - `{f(x, y)}` ⋪ `f(y, x)` ? Safe
2. Suppress triggers that could lead to matching loops
3. Report information to the user

Overly enthusiastic loop suppression causes a loss of expressive power

- Cycle detection acts on a full quantifier, while loops often only involve parts of it:

$$\forall\ x\ \{??\} \cdot P(x) \wedge (Q(x) \Longrightarrow P(x+1))$$

- Suppressing loops costs us too much expressiveness: we don't learn $P(x)$ anymore!

Splitting quantifiers regains some expressiveness

We extended Dafny to split quantifiers before checking for loops:

$$\forall\ x\ \{Q(x)\} \cdot P(x) \wedge (Q(x) \implies P(x+1)) \quad \longmapsto \quad \forall\ x\ \{P(x)\} \cdot P(x)$$
$$\forall\ x\ \{Q(x)\} \cdot Q(x) \implies P(x+1)$$

- Each quantifier gets its own triggers.
- This fixes some of our issues, but we lose a different type of expressiveness: learning $Q(x)$ doesn't teach us $P(x)$ anymore!

Triggers sharing further recovers expressive power

- Triggers do not need to appear in the body of a quantifier.
- Dafny can share triggers across all terms of a split quantifier:

$$\forall\ x\ \{P(x)\}\ \{Q(x)\} \cdot P(x)$$
$$\forall\ x\ \ \ \ \ \ \{Q(x)\} \cdot Q(x) \Longrightarrow P(x+1)$$

Overall variance results

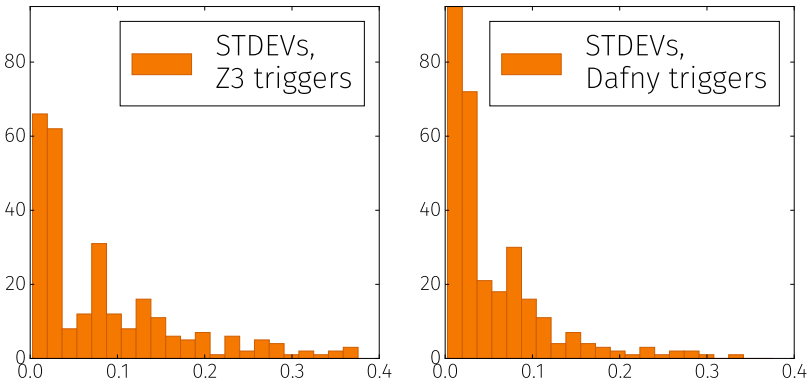

Figure : Standard deviations of single-test running times across 10 runs of the test suite

- Using Dafny-generated triggers improves variability in small ways across most of the test suite.
- The effect on most tests is small; some tests do benefit significantly.

Zooming in

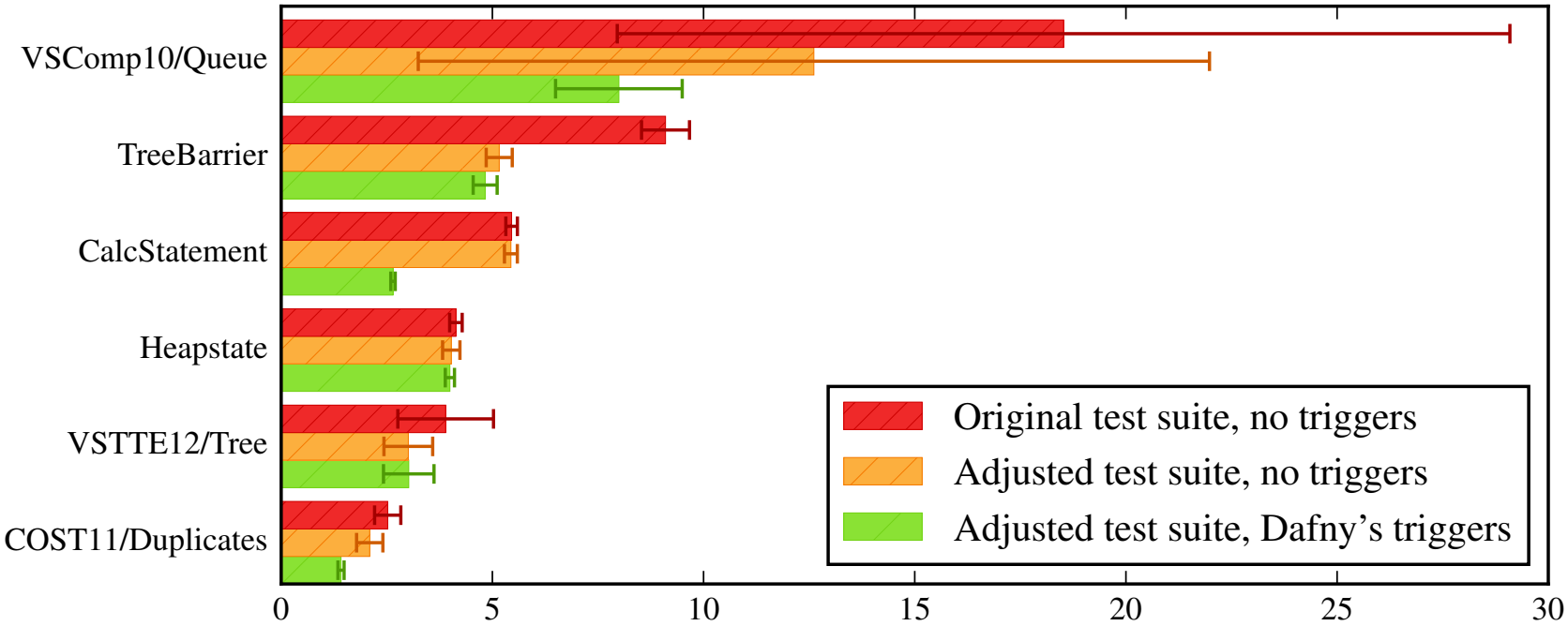

Figure : Verification times in seconds for six example programs taken from Dafny's test suite

IronFleet RSL

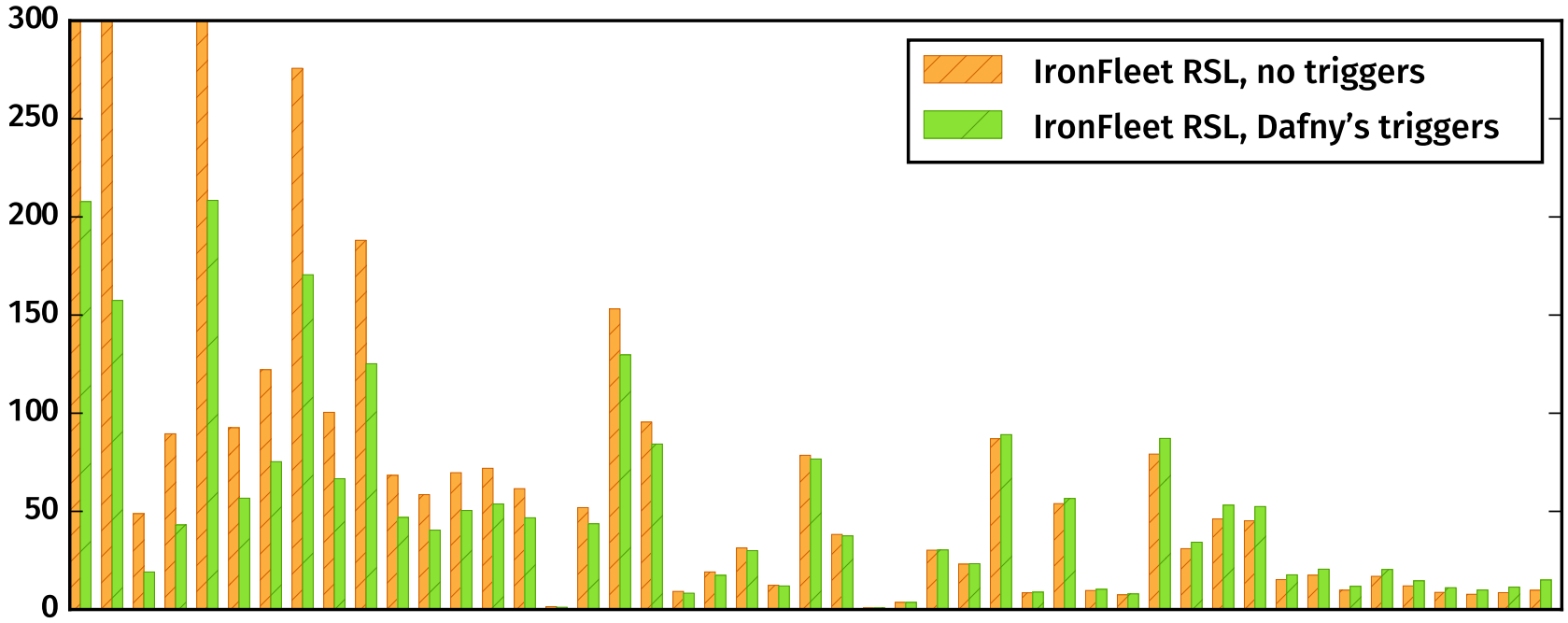

Figure : Verification times in seconds for the 48 programs composing the implementation layer of IronRSL

Usability results

- Dafny now picks triggers and reports them directly in the editor
- Generating triggers avoids parasitic terms and spurious matches
- New visualization tools help with diagnosing issues and understanding the verification process

Conclusion

- Trigger generation, quantifier splitting, and matching loop elimination offer new, exciting opportunities to improve the performance and predictability of tools based on SMT solvers.
- Efficient visualizations can yield new insight into surprising verifier behaviors.

Next steps

- Check out the all new Dafny (on GitHub! MIT-licensed!):

  https://github.com/Microsoft/dafny

- Install boogie-friends in Emacs

  https://github.com/boogie-org/boogie-friends/

  (includes `dafny-mode`, `boogie-mode`, and `z3-smt2-mode`)

- Try the new axiom profiler

  (which I'll have to email to you for now)

- Talk to me!

clement@pit-claudel.fr

http://pit-claudel.fr/clement/