Search space reduction for Dafny tactics
Vytautas Tumas, supervised by Gudmund Grov
School of Mathematical & Computer Sciences

Introduction
The aim of the project is to identify performance bottlenecks in the Tacny tool, and provide optimisation solutions.

Traditional Program Verification
Traditionally verification tasks were carried out with Interactive Theorem Provers. They allow users to define and reason about mathematical theories. The prover is guided with proof scripts called tactics.

On the other side of the spectrum there Satisfiability Modulo Theories solvers, which offer automatic program verification. One such solver is the Z3 SMT Solver, that is specialized in verifying software. It the underlying solver for the programming language and verifier Dafny.

Programming language and verifier Dafny
- Imperative, object oriented programming language.
- Supports functions without side effects.
- Designed for program verification.
- Supports specification constructs to assert facts about the program.

The good:
- Easier to learn, than traditional verification tools.
- Intuitive syntax.
- Prove program functional correctness.

The bad:
- Proving programs is repetitive, time consuming task.
- Impossible to reuse proofs.

Tacny
- Extends Dafny with high-level proof encodings: tactics
- Generates concrete proofs from the abstractions.

The good:
- A single tactic can generate multiple proofs.
- Improves code readability.
- Nonintrusive syntax.

The bad:
- Slow execution time.
- High memory usage.

Tactic Application

![Proof abstraction (tactic)]

```
tactic CasePerm(v: Element)
{
  cases (v) {
    tvar v = merge{
      variables(),
    }
    tvar l :: l in lemmas():
    perm(l, v);
    perm(l, v);
  }
}
```

![Tactic application]

```
lemma AssignConst(a: aexp, s: state)
requires Total(s)
ensures evalAssignConst(a, s) == eval(a, s)
{  
  CasePerm(a);
}
```

![Generated proof]

```
lemma AssignConst(a: aexp, s: state)
requires Total(s)
ensures evalAssignConst(a, s) == eval(a, s)
{  
  match (a) {
    case N(a) =>
    case V(a) =>
    case Plus(a, s1) =>
    AssignConst(a0, s1);
    AssignConst(a1, s1);
  }
}
```

Tacny Analysis
The analysis of the original tool revealed four performance bottlenecks:
- The size of the generated search space. Majority of the generated solutions are invalid Dafny programs.
- The search space is generated eagerly. The tool first generates the search space and then searches for the correct solution.
- Inefficient search strategies used to find the solution. Tacny uses Breadth-First-Search Strategy, however it may incur higher memory usage.
- Sequential tactic resolution. The tactic applications are resolved one at a time.

Optimization
Developed three optimizations:
- A type lookup table to ensure generated solutions are type correct.
- Lazy search space generation to reduce the search space, and memory usage.
- Support for multiple search strategies, to improve the performance of individual tactic and implement Depth-First-Search strategy.

Evaluation
The ■ shape refers to the unoptimised tool, and the ● shape indicates the tool running all optimizations.
- The introduced type lookup table has greatly reduced the size of the generated search space, thus improved the execution time and memory usage.
- The Depth-First-Search strategy provides marginally improved execution time than Breadth-First-Search strategy.
- The lazy evaluation had the greatest impact on the memory usage, it does not allow invalid solutions to accumulate, thus only few solutions are held in memory.

Conclusion and Future Work
The optimisations greatly improved execution time and reduced memory usage, and enabled further optimisations:
- The most important is the parallel tactic execution, as it should further reduce execution time.
- Explore heuristic search algorithms such as A*.
- Develop richer tactics.