Proving Program Equivalence in Dafny

Nathaniel Victor, Dragana Milovancevic, and Sophia Drossopoulou

Imperial College London

1 Introduction

Program verifiers such as Dafny [7] and KeY [2] allow developers to write proofs of program equivalence, however, they require a significant level of user assistance to complete proofs. Recently, many equivalence checkers are being developed to help make the development of program equivalence proofs easier [6, 8, 4, 5].

In this paper, we present Equidafny, our new equivalence checker built on top of Dafny. Equidafny builds on experience from the current state-of-the-art checkers, and improves over Dafny by decreasing the amount of required user assistance.

2 Design Overview

We present the overview of EquiDafny on an example of two equivalent functions: `insertSortedM` and `insertSorted1`. The first function uses the helper function `insertM`:

```
function insertSortedM<Seed>(seed: Seed, next: Seed -> (Seed, int),
                             count: int, xs: List<int>): List<int>
  {
  if (count <= 0) then xs
  else
    var (nxtS, t) := next(seed);
    insertSortedM(nxtS, next, count - 1, insertM(xs, t))
}

function insertM(xs: List<int>, t: int): List<int>
  decreases xs {
  match xs {
    case Nil => Cons(t, Nil)
    case Cons(hd, tl) =>
      if (t <= hd) then Cons(t, xs) else Cons(hd, insertM(tl, t))
  }
}
```

The second function uses the same structure, with the helper function `insert1`:

```
function insertSorted1<Seed>(seed: Seed, next: Seed -> (Seed, int),
                             xs: List<int>, count: int): List<int>
  decreases count {
```

```
  if (!(count <= 0)) then
    var (nxtS, t) := next(seed);
    insertSorted1(nxtS, next, insert1(t, xs), count - 1)
  else xs
}

function insert1(t: int, xs: List<int>): List<int>
  decreases xs {
  match xs {
    case Cons(hd, tl) =>
      if (!(t <= hd)) then Cons(hd, insert1(t, tl)) else Cons(t, xs)
    case Nil => Cons(t, Nil)
  }
}
```

The user can create an equivalence lemma for this example:

```
lemma eqInsertSorted<Seed>(seed: Seed, next: Seed -> (Seed, int),
                           xs: List<int>, count: int)
  ensures insertSorted1(seed, next, xs, count) ==
            insertSortedM(seed, next, count, xs)
{}
```

However, the Dafny verifier is unable to prove this lemma automatically.

This example illustrates three main challenges here when attempting to prove the equivalence lemma.

First: Generate and call helper lemmas for helper functions. The `insertSortedM` and `insertSorted1` functions both use the helper functions `insertM` and `insert1` in the same way, therefore, the Dafny verifier requires an equivalence lemma between these functions. To resolve this, EquiDafny creates and calls a helper equivalence lemma `eqInsert`, under the condition `count > 0` and with the correct arguments.

Second: Provide a variant. EquiDafny provides a suitable variant for proving termination (`count`), copied from the `InsertSorted1` function.

Third: Explicitly stating the induction scheme. EquiDafny provides a recursive call to the `eqInsertSorted` lemma to complete the proof.

Thus, EquiDafny generates the following:

```
lemma eqInsert(t: int, xs: List<int>)
  ensures (insert1(t, xs) == insertM(xs, t))
{}

lemma eqInsertSorted<Seed>(seed: Seed, next: Seed -> (Seed, int),
                           xs: List<int>, count: int)
  ensures (insertSorted1(seed, next, xs, count)
           == insertSortedM(seed, next, count, xs))
  decreases count
```

```
{
  if count > 0 {
    var (nxtS, t) := next(seed);
    eqInsertSorted(nxtS, next, insert1(t, xs), count - 1);
    eqInsert(t, xs);
  }
}
```

3 Function Merging

The main algorithm EquiDafny uses is called 'Function Merging'. By exploiting similarities between the structures of function bodies, these structures can be split into their constituent parts and recursively 'merged' together. This algorithm has two outputs. The first is a set of mappings between the parameters of one function to the parameters of the other. This is used to describe how the arguments passed into one function in an equivalence lemma must be permuted to form the arguments to the second function. The second output is the body of each lemma. In our example, this involves the call to the helper equivalence lemma `eqInsert`. Additionally, the algorithm involves merging function call arguments correctly and merging mutually recursive functions.

4 Evaluation

We evaluated EquiDafny against the base Dafny verifier and the Stainless equivalence checker. We considered the set of program equivalence tests from the Stainless verifier [1], from the related work on measure transfer [9] and from the EqBench dataset [3] and converted them to Dafny. We additionally created a few new tests. The entire dataset is publicly available at:
https://github.com/NVictor2004/dafny-equivalence-benchmarks.

The following table provides metrics describing how complex these tests are. These metrics are the number of lines of code in each test file and the number of lemmas needed to verify each test.

Original Source	Lines of Code			Number of Lemmas		
	Min	Mean	Max	Min	Mean	Max
Stainless equivalence tests	2	34.42	91	1	1.81	6
EqBench dataset	18	34.04	84	1	1.46	4
Measure transfer artifact	18	41.43	86	1	1.60	2
Self-Written	12	34.50	54	1	1.64	2

The results of our evaluation are shown below. Thus, EquiDafny improves the base Dafny verifier and is almost on par with the state-of-the-art checker Stainless.

Original Source	Number of Tests	Tools		
		Dafny	EquiDafny	Stainless
Stainless equivalence tests	53	21	43	52
EqBench dataset	52	14	16	34
Measure Transfer Artifact	7	0	2	2
Self-Written	14	2	9	5
Total	126	37	70	93

Stainless can verify significantly more tests than EquiDafny can, however, EquiDafny can verify several tests that Stainless cannot. For example, Stainless cannot verify tests that require type transformations.

5 Ongoing and Future Work

This section outlines how EquiDafny will be improved in the future.

Datatype Transformations In order to prove program equivalence, tools such as Stainless require equivalent programs to have identical type signatures. For example, changing `next: Seed -> (Seed, int)` to `next: Seed -> (int, Seed)` in one of the functions cannot be treated by Stainless. In contrast, EquDafny has limited support for proving function equivalence even when functions operate on different types. In the future, we plan to extend EquDafny to support more complex type transformations.

User Interface EquiDafny is currently running as a command line tool, but Dafny can also run in a VSCode plugin. We plan to implement a new plugin for EquiDafny to be used within VSCode.

References

1. Stainless/frontends/benchmarks/equivalence at main · epfl-lara/stainless. https://github.com/epfl-lara/stainless/tree/main/frontends/benchmarks/equivalence
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, http://dx.doi.org/10.1007/978-3-319-49812-6
3. Badihi, S., Li, Y., Rubin, J.: EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). pp. 610–614. IEEE, Madrid, Spain (May 2021). https://doi.org/10.1109/MSR52588.2021.00084
4. Felsing, D., Grebing, S., Klebanov, V., Rummer, P., Ulbrich, M.: Automating regression verification. pp. 928–951 (Sep 2014). https://doi.org/10.1145/2642937.2642987
5. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Software Testing, Verification and Reliability **23**, 241–258 (2013). https://doi.org/10.1002/stvr.1472

6. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification, vol. 7358, pp. 712–717. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_54
7. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, vol. 6355, pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
8. Milovančević, D., Kunčak, V.: Proving and Disproving Equivalence of Functional Programming Assignments. Proceedings of the ACM on Programming Languages **7**(PLDI), 928–951 (Jun 2023). https://doi.org/10.1145/3591258
9. Milovančević, D., Fuhs, C., Bucev, M., Kunčak, V.: Proving Termination via Measure Transfer in Equivalence Checking. In: Kosmatov, N., Kovács, L. (eds.) Integrated Formal Methods. pp. 75–84. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-76554-4_5