Proving Termination via Measure Transfer in Equivalence Checking

Dragana Milovančević[1]✉[0009−0003−0795−881X] dragana.milovancevic@epfl.ch, Carsten Fuhs[2]✉[0009−0007−3697−4383] c.fuhs@bbk.ac.uk, Mario Bucev[1]✉mario.bucev@epfl.ch, and Viktor Kunčak[1]✉[0000−0001−7044−9522] viktor.kuncak@epfl.ch

[1] EPFL, Station 14, CH-1015 Lausanne, Switzerland
[2] Birkbeck, University of London, United Kingdom

Abstract. Program verification can benefit from proofs with varied induction schemas. A natural class of induction schemas, functional induction, consists of those derived from definitions of functions. For such inductive proofs to be sound, it is necessary to establish that the functions terminate, which is a challenging problem on its own. In this paper, we consider termination in the context of equivalence checking of a candidate program against a provably terminating reference program annotated with termination measures. Using equivalence checking, our approach automatically matches function calls in the reference and candidate programs and proves termination by transferring measures from a measure-annotated program to one without annotations. We evaluate this approach on existing and newly written termination benchmarks, as well as on exercises in programming courses. Our evaluation corpus comprises around 10K lines of code. We show empirically that the termination measures of reference programs often successfully prove the termination of equivalent candidate programs, ensuring the soundness of inductive reasoning in a fully automated manner.

Keywords: Equivalence checking · Termination analysis · Termination measures.

1 Introduction

Termination is a prototypical liveness property of programs; verifying program termination is a long-standing problem in computing. Termination is also an interesting property for practical program verification because it is *implicit*, in the sense that it does not require developers to write specifications. It is important because non-interactive non-terminating computations fail to deliver any useful functionality. Furthermore, programs can be used to encode proofs and proof hints through ghost state or proof irrelevance. Termination of such programs is crucial for the soundness of all reasoning, even when reasoning applies to safety.

In this paper, we focus on termination analysis in the context of program equivalence. Equivalence checking has applications in proving program optimizations [27], regression verification [9, 28], refactoring [19], and automated grading [21]. However, while termination is often a prerequisite for the soundness of an equivalence proof, many existing approaches put termination analysis aside and leave it as the responsibility of end users [7,9]. Approaches that omit termination checks can result in unsound equivalence proofs, leading to incorrect conclusions. To illustrate this problem, we tried feeding the following (non-equivalent) functions to the REVE equivalence checker [9], and got the following verdict: “The programs have been proved equivalent”! As the authors of REVE state in [9], their approach indeed proves equivalence under the assumption of termination, which does not hold here. On the other hand, in tools that do integrate equivalence checking with termination analysis, termination checks may fail systematically [21]. As a result, such tools often need many manual annotations, reducing the opportunities for fully automated deployment.

```
int foo(int x) {
  int r;
  if (x == 1) r = 1;
  else r = foo(x);
  return r;
}
```

```
int foo(int x) {
  int r;
  if (x == 1) r = 1;
  else r = 2;
  return r;
}
```

Researchers have built dedicated tools for automated termination checking [2, 6, 10, 15, 16, 30]. Termination analysis is also integrated in proof assistants [13,24] and program verifiers such as Dafny [18] or Stainless [17], by means of synthesizing and verifying *termination measures*, also known as *ranking functions*. Termination checking in Stainless was in part carried over from Leon [29], along with support for generic types and quantifiers [31]. Subsequent work introduced a foundational type system that enforces termination [12].

Tools like Stainless are good at *verifying* that a given termination measure for a program is a valid termination proof. However, they are less good at *synthesizing* termination measures. Those limitations of termination analysis in Stainless are evidenced in recent case studies, including the LongMap proof [5], which modifies the implementation to add loop counters to prove termination, and the QOI proof [3], with 23 measure annotations for 313 lines of implementation.

In this paper, we consider the automatic transfer of termination measures between potentially-equivalent programs to facilitate equivalence checking. We extend the equivalence checking functionality of the Stainless verifier [21] to perform measure transfer along with its automated equivalence proof generation. We build on Stainless for the reasons discussed above, with an eye on improving automation of its equivalence checking component. We evaluate measure transfer on termination and equivalence checking benchmarks and on student assignments. We find that in most cases where Stainless can prove program equivalence, measure transfer results in a successful termination proof, effectively eliminating the need for manual annotations. In general, one should expect that equivalent programs written by different developers will require a broad selection of different termination measures. Our insight is that, in a practical case study on student assignments, this diversity of measures usually does not arise. This insight leads to a simple automated termination checking approach.

```
def finite(s: Stream): Boolean =
  s match
    case SCons(_, tf, sz) if tf().rank ≥ sz ⇒
      false
    case SCons(_, tf, sz) ⇒ finite(tf())
    case _ ⇒ true
```

(a) Initial implementation. Stainless fails to infer the measure and cannot prove termination.

```
def finite(stream: Stream): Boolean =
  stream match
    case SCons(_, tfun, sz) ⇒
      val tail = tfun()
      tail.rank < sz && finite(tail)
    case SNil() ⇒ true
```

(b) Refactored implementation. Stainless fails to prove termination.

```
def finite(s: Stream): Boolean =
  decreases(s.rank) //given
  s match
    case SCons(_, tf, sz) if tf().rank ≥ sz ⇒
      false
    case SCons(_, tf, sz) ⇒ finite(tf())
    case _ ⇒ true
```

(c) The **decreases** annotation, provided by the user, specifies the measure that decreases in each recursive call. With the help of **decreases** annotation, Stainless succeeds at proving termination.

```
def finite(stream: Stream): Boolean =
  decreases(stream.rank) //inferred
  stream match
    case SCons(_, tfun, sz) ⇒
      val tail = tfun()
      tail.rank < sz && finite(tail)
    case SNil() ⇒ true
```

(d) Automated porting of the **decreases** annotation. Stainless succeeds at proving termination and equivalence to finite from Figure 1c.

Fig. 1: Refactoring and measure transfer. Manually inserted annotations (c) are marked in red; annotations inferred via measure transfer (d) are marked in blue.

An extended version of this paper is available as a technical report [22] under a CC BY-NC-ND license.

2 Illustrative Example

In this section, we illustrate our approach on an example of program refactoring. We consider functions operating on user-defined streams:

```
sealed abstract class Stream
  def rank = {
    this match
      case SCons(_, _, sz) if (sz > 0) ⇒ sz
      case _ ⇒ BigInt(0)
  } ensuring(_ ≥ 0)
case class SCons(x: BigInt, tailFun: () ⇒ Stream, sz: BigInt) extends Stream
case class SNil() extends Stream
```

Termination analysis of a similar encoding of streams in Stainless was previously discussed in detail [12]. Here, we consider function finite, which defines a sufficient condition that the input stream is finite, according to the provided ranked values.

Figure 1 shows two implementations of function finite: the initial implementation (1a) and the refactored implementation (1b). We run Stainless to prove termination of the initial implementation (1a), and we encounter a timeout. We thus fall back on inserting manual measure annotations (1c). The **decreases** clause specifies that the function terminates because the rank of the input stream decreases at each recursive call. To prove the equivalence of the two implementations of finite, Stainless requires that both functions terminate. However, when running Stainless to prove termination of the refactored implementation (1b), once again, we obtain a timeout. Rather than providing another manual measure annotation for the refactored program, we utilize the equivalence checking component to automatically perform measure transfer (1d). As a result, the system is able to prove termination, which completes the equivalence proof.

We found this approach particularly useful for larger programs, to automatically map and transfer measures for inner functions (Appendix A in our technical report [22]). We identify further applications in automated grading of programming assignments, with several reference programs with different termination measures (Appendix B in our technical report [22]).

3 Measure Transfer

In this section, we explain how we use measure transfer from potentially-equivalent programs as a heuristic to speculate termination measures.

Terminology. In our context, a *measure* is a lexicographic combination of function(s) from function arguments to natural numbers. A measure m proves the termination of a recursive function F if there is a decrease of the value of m for each recursive call. In this case, m is also called a *termination measure* for F. In Stainless, measures are provided as annotations for a function using the **decreases** keyword. The annotations consist of (lexicographic combinations of) Scala expressions.

We use the term *measure transfer* to refer to the general process of taking a termination measure m for a function M and conjecturing that m (or a permutation of m) is also a termination measure for a function F. This transfer is guided by a partial equivalence proof of M and F as a heuristic.

Algorithm. We consider a setting with one or more single-function reference programs, annotated with termination measures, and one or more single-function candidate programs, without measure annotations. For each candidate program, our algorithm is as follows:

1. Given: Reference programs $M_1, M_2, \ldots, M_n$ with their respective termination measures $m_1, m_2, \ldots, m_n$, proven terminating, and a candidate program under analysis P.
2. Check whether there exists M_i provably equivalent to P (under the assumption that P is terminating) such that m_i is also a termination measure for P (proving the assumption).

If there are multiple reference programs that are provably equivalent to P (modulo the termination of P), we consider them all until we find a measure m_i that proves the termination of P (or we exhaust all options). If such a measure m_i exists, this concludes the equivalence proof.

The potential equivalence of M_i and P provides the motivation for trying m_i as a candidate termination measure, but there is no guarantee that m_i should be a termination measure for P. However, our experiments (Section 4) show that this is often the case in our benchmarks.

Auxiliary Functions. Consider next a setting where programs consist of one or more auxiliary functions. Termination proofs in Stainless are performed separately for each function as entry point [12]. For programs comprising multiple functions, we identify pairs of potentially-equivalent functions for measure transfer. For each such pair, our algorithm is as follows:

1. Given: A reference function M with termination measure m, proven terminating, and a candidate function under analysis P, proven equivalent to M *for some argument permutation* under the assumption that P is terminating,
2. Check whether m is also a termination measure for P, for the same argument permutation (proving the assumption).

Search for potentially-equivalent auxiliary functions and corresponding argument permutations is inherited from the equivalence checking component of Stainless; it is based on type- and test-directed search [21]. For an example of measure transfer for auxiliary functions, see Appendix A of our report [22].

4 Evaluation

We implement measure transfer as a new mode for termination proving as part of equivalence checking in the Stainless verifier, as an alternative to existing measure inference. We evaluate measure transfer on termination and equivalence benchmarks, as well as on programming assignments. For each run, we set a 10s timeout for Z3 solver queries.

Benchmarks. Table 1 presents our collection of existing and newly written benchmarks for termination and equivalence checking. We consider programs with state, user-defined types, type parameters, as well as helper functions and higher-order functions. Each benchmark contains two equivalent Scala programs: one reference program, annotated with termination measures, and one refactored candidate program, without any measure annotations.

Table 2 describes benchmarks from programming courses [20, 21]. Each benchmark contains one or more reference solutions annotated with termination measures, and equivalent student submissions with no measure annotations, where automated measure inference fails.

The source code of all our benchmarks is publicly available together with our implementation in Stainless [23].

Table 1: Evaluation results. LOC: total number of lines of code. F and D: number of functions and number of measure (decreases) annotations in reference program, respectively. I and T: outcome of equivalence checking when using measure inference and measure transfer, respectively (✓ indicates success and ✗ indicates failure). IT and TT: total time for equivalence checking when using measure inference and measure transfer, respectively.

Name	LOC	F	D	I	IT[s]	T	TT[s]	src
AdjList	32	2	1	✓	26.12	✓	23.74	New
ArrayContent	12	1	1	✓	12.85	✓	13.32	New
ArrayHeap	58	4	1	✓	27.47	✓	26.19	New
ArrayInc	15	2	1	✗	N/A	✓	18.12	New
Boardgame	293	8	3	✗	N/A	✓	1186.6	New
FiniteStreams	28	1	1	✗	N/A	✓	16.07	[17]
MaxHeapify	51	3	1	✗	N/A	✓	15.12	New
Partial	27	4	1	✗	N/A	✓	15.80	[17]
SortedArray	26	2	1	✓	15.18	✓	13.03	New
Valid2DLen	17	1	1	✗	N/A	✓	19.10	New

Table 2: Further evaluation results, on programming assignments. LOC: average number of lines of code per program. F and D: average number of function definitions and average number of measure annotations per program, respectively. R: number of reference programs. S: number of submissions. I and T: number of submissions with successful equivalence proof by measure inference and measure transfer, respectively.

Name	LOC	F	D	R	S	I	T	src
gcd	9	1	1	2	41	0	22	[20]
formula	59	2	1	1	37	0	27	[21]
prime	21	4	2	2	22	0	5	[20]
sigma	10	1	1	3	704	0	678	[21]

Results. Measure transfer succeeds for all benchmarks in Table 1, including 6 benchmarks where the type-based measure inference in Stainless [12] fails. Furthermore, for benchmarks where measure inference succeeds, measure transfer typically reduces the processing time. Occasionally, the overhead of measure transfer transformations results in a slight time increase (e.g., in the smallest ArrayContent benchmark).

Out of 3 benchmarks with multiple reference programs (Table 2), only gcd has different termination measures (shown in Appendix B of our report [22]). In the sigma benchmark, measure transfer succeeds for 678 submissions, out of 704 submissions that previously required manual annotations due to limitations of measure inference [21]. The 26 submissions where measure transfer fails are due to either equivalence proof failures (11) or due to introducing inner functions that exist only in the candidate submission (15). In the formula benchmark, for 37 submissions where measure inference fails, the evaluation in [21] uses manual annotations to prove termination of 25 submissions (for the remaining submissions, the manual annotator did not find a termination measure). In contrast, measure transfer automatically proves correctness of 27 submissions. In the prime benchmark, we encounter a submission that, when manually annotated, passes termination checks and is equivalent to one reference solution. However, the measure transfer fails, because the inner function's measure in the reference solutions gives a negative measure when transferred to the inner function of the submission.

5 Related Work

Our approach is related to ACL2's `defunT` macro [14], which searches for termination measures in a database of already-proved termination theorems. Similarly, we use reference programs to search for termination measures. Both `defunT` and our work are instances of proof transfer [8] between related theorems. In our case, the theorems have the form "function f terminates for all inputs", and we use equivalence as a heuristic in choosing candidate termination measures. More generally, this line of work is in the space of *proof repair* [25], where work on verified programs so far seems to have focused mainly on partial correctness [11,26], with the above exceptions.

We address the automatic synthesis of **decreases** annotations as termination measures, which are also used in JML-like settings [4]. These termination measures are (lexicographic combinations of) functions to $\mathbb{N}$ and cover an important class of termination proofs by ranking functions. Our work is related to the coupling of the tools COSTA and KeY [1]. Given a Java program, COSTA finds a termination measure, used by KeY to independently verify termination. However, COSTA needs to solve non-trivial constraint problems and considers only specific shapes for the measures. Our approach uses significantly less search and is not restricted to measures of a specific shape.

6 Conclusions

We have presented challenging examples in termination analysis in the context of equivalence proofs, and have shown how we can address them using a technique as simple as measure transfer. Our evaluation showed that measure transfer is effective in practice: it provided significant improvement over the automated measure inference in Stainless (including a speed-up in processing time), and sometimes an improvement over manual annotations. This shows that, for applications such as automated grading, where standard classes of termination measures may fail, measure transfer can lead to improvements for automation and applicability. In the future, we will consider more complex measure transformations, including transfer of termination lemmas.

Acknowledgments. This publication is based on collaboration within the COST Action CA20111 - European Research Network on Formal Proofs (EuroProofNet), supported by COST (European Cooperation in Science and Technology, www.cost.eu), as well as the EPFL Doc.Mobility Grant "Termination Checking for Sound Equivalence Proofs of Real-World Programs". We thank the anonymous reviewers for their helpful feedback. We thank Ioana Jianu for helping the development of array benchmarks.

Availability of Data and Software Stainless is under active development and is available at [17]. The complete data set and instructions for reproducing the results from this paper are available in an open access Zenodo repository [23].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: A formal verification framework for static analysis - as well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. Softw. Syst. Model. **15**(4), 987–1012 (2016). `https://doi.org/10.1007/S10270-015-0476-Y`, `https://doi.org/10.1007/s10270-015-0476-y`
2. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 387–393. Springer (2016). `https://doi.org/10.1007/978-3-662-49674-9_22`, `https://doi.org/10.1007/978-3-662-49674-9_22`
3. Bucev, M., Kunčak, V.: Formally verified quite OK image format. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 343–348. IEEE (2022). `https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_41`, `https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_41`
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. **7**(3), 212–232 (2005). `https://doi.org/10.1007/S10009-004-0167-4`, `https://doi.org/10.1007/s10009-004-0167-4`
5. Chassot, S., Kunčak, V.: Verifying a realistic mutable hash table - case study (short paper). In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14739, pp. 304–314. Springer (2024). `https://doi.org/10.1007/978-3-031-63498-7_18`, `https://doi.org/10.1007/978-3-031-63498-7_18`
6. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). `https://doi.org/10.1145/3192366.3192405`, `https://doi.org/10.1145/3192366.3192405`
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Hipspec: Automating inductive proofs of program properties. In: Fleuriot, J.D., Höfner, P., McIver, A., Smaill, A. (eds.) ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation, Manchester, UK, June 2012. EPiC Series in Computing, vol. 17, pp. 16–25. EasyChair (2012). `https://doi.org/10.29007/3qwr`
8. Cohen, C., Crance, E., Mahboubi, A.: Trocq: Proof transfer for free, with or without univalence. In: Weirich, S. (ed.) Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14576, pp. 239–268. Springer (2024). `https://doi.org/10.1007/978-3-031-57262-3_10`, `https://doi.org/10.1007/978-3-031-57262-3_10`

9. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 349–360. ASE '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642987, https://doi.org/10.1145/2642937.2642987
10. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). https://doi.org/10.1007/S10817-016-9388-Y, https://doi.org/10.1007/s10817-016-9388-y
11. Gopinathan, K., Keoliya, M., Sergey, I.: Mostly automated proof repair for verified libraries. Proc. ACM Program. Lang. **7**(PLDI), 25–49 (2023). https://doi.org/10.1145/3591221, https://doi.org/10.1145/3591221
12. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360592, https://doi.org/10.1145/3360592
13. INRIA: Functional induction in coq. https://coq.inria.fr/refman/using/libraries/funind.html (2021)
14. Kaufmann, M.: DefunT: A tool for automating termination proofs by using the community books (extended abstract). In: Goel, S., Kaufmann, M. (eds.) Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, November 5-6, 2018. EPTCS, vol. 280, pp. 161–163 (2018). https://doi.org/10.4204/EPTCS.280.12, https://doi.org/10.4204/EPTCS.280.12
15. Kop, C.: WANDA - a higher order termination tool (system description). In: Ariola, Z.M. (ed.) 5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference). LIPIcs, vol. 167, pp. 36:1–36:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPICS.FSCD.2020.36, https://doi.org/10.4230/LIPIcs.FSCD.2020.36
16. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8410, pp. 392–411. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_21, https://doi.org/10.1007/978-3-642-54833-8_21
17. LARA, E.: Stainless. https://github.com/epfl-lara/stainless (2023)
18. 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 - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
19. Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 329–339 (2021). https://doi.org/10.1109/ICST49551.2021.00045

20. Milovancevic, D., Bucev, M., Wojnarowski, M., Chassot, S., Kuncak, V.: Formal autograding in a classroom (experience report) (2024), http://infoscience.epfl.ch/record/309386
21. Milovančević, D., Kunčak, V.: Proving and disproving equivalence of functional programming assignments. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591258, https://doi.org/10.1145/3591258
22. Milovančević, D., Fuhs, C., Bucev, M., Kuncak, V.: Proving Termination via Measure Transfer in Equivalence Checking (Extended Version). Tech. rep., EPFL (sep 2024)
23. Milovančević, D., Fuhs, C., Bucev, M., Kunčak, V.: Proving Termination via Measure Transfer in Equivalence Checking (Artifact) (Sep 2024). https://doi.org/10.5281/zenodo.13787855, https://doi.org/10.5281/zenodo.13787855
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002), https://doi.org/10.1007/3-540-45949-9
25. Ringer, T.: Proof Repair. Ph.D. thesis, University of Washington, USA (2021), https://hdl.handle.net/1773/47429
26. Ringer, T., Porter, R., Yazdani, N., Leo, J., Grossman, D.: Proof repair across type equivalences. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 112–127. ACM (2021). https://doi.org/10.1145/3453483.3454033, https://doi.org/10.1145/3453483.3454033
27. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 391–406. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509509, https://doi.org/10.1145/2509136.2509509
28. Strichman, O., Godlin, B.: Regression verification - A practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 496–501. Springer (2005). https://doi.org/10.1007/978-3-540-69149-5_54
29. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6887, pp. 298–315. Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_23, https://doi.org/10.1007/978-3-642-23702-7_23
30. Urban, C.: FuncTion: An abstract domain functor for termination - (competition contribution). In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 464–466. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_46, https://doi.org/10.1007/978-3-662-46681-0_46
31. Voirol, N.: Termination Analysis in a Higher-Order Functional Context. Master's thesis, EPFL (2023), http://infoscience.epfl.ch/record/311772