Experience Report: Type-Driven Development of Certified Tree Algorithms in Coq

Reynald Affeldt[1], Jacques Garrigue[2], Xuanrui Qi[2,3], and Kazunari Tanaka[2]

[1]National Institute of Advanced Industrial Science and Technology, Japan
[2]Graduate School of Mathematics, Nagoya University, Japan
[3]Department of Computer Science, Tufts University, USA

Abstract

This talk is a report on, as well as a defense of the usefulness of dependent types for developing provably correct programs. We believe that dependent types—and dependently-typed programming in Coq in particular—could allow for faster and safer development. There are already several accounts about the utility of dependent types in practical program development, and in real-world applications. Here we add to these accounts by outlining our experience in developing tree algorithms for succinct data structures and proving them with the help of dependent types.

1 Motivation: Formalization of Red-black Trees

Ever since its invention, dependently-typed programming has been mostly of theoretical interest. However, recent work has showed that dependently-typed programming could also be useful in practical programming, especially when the program being developed has strict security requirements or complex invariants [4, 13]. Here, we present another case where dependently-typed programming was used to verify such algorithms—in this case, an algorithm that looks innocuous but in fact has rather byzantine invariants.

The Problem Implementing deletion for purely functional red-black trees was long known to be a tricky problem. Kahrs and Germane and Might have given relatively simple algorithms [6, 7], but neither comes with a formal proof, and both proved difficult to refactor for our purposes. Also, we could not easily reuse existing Coq formalizations of red-black trees (such as [2]) because our application (to succinct data structures[1] [1]) required nodes decorated with meta-data. We started with a direct translation of Kahrs' algorithm to Coq, but soon we found ourselves unconvinced of the correctness of our implementation and thus further progress was stalled.

Can Types Help? Type-driven development has been advocated in the functional programming community for a long time, but does it apply as well when we want to fully prove an algorithm? Dependent types in Coq allow one to enforce invariants of data-structures and algorithms within their types. It is tempting to use them to implement an algorithm together with its proof, possibly refining the invariants on the go. We will compare that with the more traditional approach of using only ML-style polymorphic types, and writing external lemmas to state the desired correctness properties (this is the approach we used in our main development [1]).

For example, let `btree` be the type of colored binary trees. We can implement balancing in direct-style along the following lines (see [1, file `dynamic_redblack.v`] for all details):

[1]Also referred to as "compact data structures" in some sources.

```
Definition balanceL col (l r : dtree) dl : dtree := match col with
  | Red => Bnode Red l dl r
  | Black => match l with
             | Bnode Red (Bnode Red a da b) dab c =>
               Bnode Red (Bnode Black a da b) dab (Bnode Black c (subD dl dab) r)
             | ...
             end end.
```

and prove the properties of balancing afterwards, e.g., the fact that balancing does not change the level-order traversal of nodes:

```
Lemma dflatten_balanceL c l r d : dflatten (balanceL c l r d) = dflatten l ++ dflatten r.
```

We also prove lemmas about two orthogonal sets of invariants: the shape invariants which guarantee that our trees are well-formed red-black trees, and the data integrity invariants which state that the meta-data in our data representation correctly captures properties of the represented data.

Using dependent types, we can encode the tree invariants inside their types, and implement balancing as a function returning not only the result but also a proof that it preserves the level-order of nodes:

```
Definition balanceL {nl ml d cl cr nr mr} p (l : near_tree nl ml d cl) (r : tree nr mr d cr) :
  color_ok p (fix_color l) -> color_ok p cr ->
  {tr : near_tree (nl + nr) (ml + mr) (inc_black d p) p | dflatteni tr = dflatteni l ++ dflatten r}.
```

Here `near_tree` and `tree` encode two variants of the red-black invariants, together with the data integrity invariants, so that this type really encodes the full specification of balancing.

To define this function, Coq offers two possibilities for complete definitions: tactics or the `Program` [9] command, both guiding interactive development through the typing context.

2 Red-black Trees Formalized using Dependent Types

In this talk, we report on the implementation using dependent types of a library of functions for red-black trees. We have implemented functions for balancing, insertion, and deletion (as well as other functions more specific to succinct data structures) using the three approaches (direct-style, dependent types with tactics, dependent types with `Program`). We discuss in particular the following points:

- We compare the direct-style approach for formalizing red-black trees with approaches using dependent types. The direct-style approach is documented elsewhere [1]. The dependent type-style approach is documented in [1, file `dynamic_dependent_tactic.v`] for the approach using Coq tactics and in [1, file `dynamic_dependent_program.v`] for the approach using the `Program` command. Overall, developing using tactics is not as painful as one may think, as one has a lot of information from the types to almost fully guide the development process. The version using `Program`, on the other hand, was more problematic: at first, we ran into bugs and limitations of the system such that all our attempts at defining `balanceL` failed. Eventually, we found out a way to write our algorithm using `Program`—namely pattern-match explicitly on every case analysis, one at a time, even when only one branch is impossible, but this unfortunately leads to extremely verbose code.

- Our effort is a substantial application of the `Program` command. For illustration, this can be appreciated by the number of obligations generated by `Program` (see the table on the right). Note that in `ddelete`, none of the goals were automatically solved because we disabled the automatic simplification of goals (using `Obligation Tactic := idtac`), since the default tactic used to simplify goals, `program_simpl`, made some goals invalid [2]. However, using `Solve All Obligations with program_simpl`, we were eventually able to solve 102 of the 161 obligations automatically.

`Program` term	generated oblig.	remaining oblig.
`Definition balanceL`	30	7
`Definition balanceR`	28	9
`Fixpoint dins`	27	20
`Fixpoint ddelete`	161	161

[2]We have also consulted with the developers of `Program` about our issues.

What did "type-driven" mean in our experiment? In practice, we started with direct-style and felt overwhelmed by the many intermediate lemmas. We realized that dependent types could help rationalize our development without disrupting the process of proving. For this purpose, `Program` looked appropriate but unfortunately turned out to be cumbersome. However, keeping dependent types and reverting to proof using tactics cleared up the path so that we could eventually get back to complete the proof using `Program`. The story is a bit different for deletion. At first, we couldn't find the correct lemmas to prove our adaptation of Kahrs' algorithm. So we developed simultaneously a deletion algorithm and its invariants using tactics. We then obtained a direct-style version by extracting the computational part of the proof, and proved it separately. We also developed a `Program` version based on the direct-style version. Moreover, for the sake of comparison, one of the authors is also working on developing the same algorithms using Coq `Equations` [10], Agda, and F* [11] [3].

3 Related Work

McBride gives a fully correct-by-construction, dependently-typed version of insertion and deletion algorithms for 2-3 trees [8], and the same approach can be easily applied to red-black trees as well. However, while McBride's algorithm is elegant and generalizable, it requires a deep understanding of many dependently-typed programming techniques, and of the fact that insertion into and deletion from balanced trees are instances of the indexed zipper construction. Although McBride's approach can also be characterized as "type-driven", it serves a different purpose, namely using dependent types to program certain algorithms in a generic way.

Weirich also gives a dependently-typed version of insertion and deletion algorithms for red-black trees [12] in dependently-typed Haskell [5]. Weirich's version is similar to Chlipala's well-known dependently-typed formalization of red-black trees in Coq [3] (although Chlipala omits the deletion operation, which is significantly harder than insertion). The earliest known attempt in developing insertion & deletion algorithms for red-black trees with type-level correctness guarantees is due to Kahrs, although the invariants enforced in Kahrs' work [7] are significantly weaker than in the other versions, due to the absence of many features essential for dependently-typed programming in Haskell at the time.

Acknowledgements We thank Matthieu Sozeau for answering questions about and discussing issues with the `Program` framework, Pierre-Évariste Dagand for pointing out the relationship between balanced tree algorithms and indexed zipper constructions and some related work, and the anonymous reviewers for feedback that improved this extended abstract. We acknowledge partial support from the JSPS KAKENHI Grant Number 18H03204.

References

[1] R. Affeldt, J. Garrigue, X. Qi, and K. Tanaka. "Proving tree algorithms for succinct data structures". In: *Proc. of ITP*. Code at `https://github.com/affeldt-aist/succinct`. Sept. 2019.

[2] A. W. Appel. "Efficient Verified Red-Black Trees". Available at `http://www.cs.princeton.edu/~appel/papers/redblack.pdf` (Coq standard library, file `MSetRBT.v`, extra modifications by P. Letouzey). Sept. 2011.

[3] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT, 2013.

[4] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J.-K. Zinzindohoué. "Implementing and Proving the TLS 1.3 Record Layer". In: *Proc. of IEEE S&P.* May 2017, pp. 463–482.

[5] R. A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". PhD thesis. University of Pennsylvania, 2016.

[6] K. Germane and M. Might. "Deletion: The curse of the red-black tree". In: *J. Funct. Program.* 24.4 (2014), pp. 423–433.

[3] In the GitHub repository: `https://github.com/xuanruiqi/dtp`.

[7] S. Kahrs. “Red-black trees with types”. In: *J. Funct. Program.* 11.4 (2001), pp. 425–432.

[8] C. McBride. “How to Keep Your Neighbours in Order”. In: *Proc. of ACM ICFP.* Aug. 2014.

[9] M. Sozeau. “Un environnement pour la programmation avec types dépendants”. PhD thesis. Université de Paris-Sud, 2008.

[10] M. Sozeau and C. Mangin. “Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019).

[11] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. “Dependent Types and Multi-monadic Effects in F*”. In: *Proc. of ACM POPL.* 2016.

[12] S. Weirich. *Examples of Dependently-Typed Programs in Haskell.* 2018. URL: `https://github.com/sweirich/dth`.

[13] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. “HACL*: A Verified Modern Cryptographic Library”. In: *Proc. of ACM CCS.* Nov. 2017, pp. 1789–1806.