Type-Driven Development of Certified Tree Algorithms

An Experience Report on Dependently-Typed Programming in Coq

Reynald Affeldt [1] Jacques Garrigue [2,4]
Xuanrui Qi [2,3] 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~~

[4]Inria Paris, France

September 8, 2019
The Coq Workshop 2019, Portland, Oregon

Our story

Dependently-typed Programming in Coq: Why and How?

The "Why" Why did we use dependent types in Coq? Under what occasions are they useful?

The "How" What is the best approach towards dependently-typed programming in Coq? How did we approach it?

Our story

Dependently-typed Programming in Coq: Why and How?

The "Why" Why did we use dependent types in Coq? Under what occasions are they useful?

The "How" What is the best approach towards dependently-typed programming in Coq? How did we approach it?

Our story

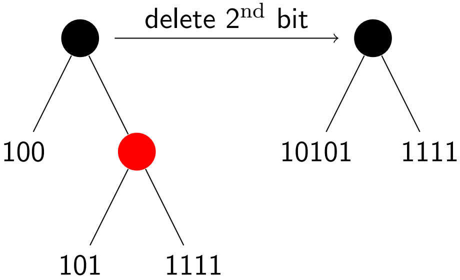
Dependently-typed Programming in Coq: Why and How?

The "Why" Why did we use dependent types in Coq? Under what occasions are they useful?

The "How" What is the best approach towards dependently-typed programming in Coq? How did we approach it?

Quick recap: what were we working on?

Bit vectors with efficient insert & delete:



- Represented using a red-black tree
- Insertion and deletion might involve inserting/deleting nodes

The motivation

Deletion from red-black trees is too hard.

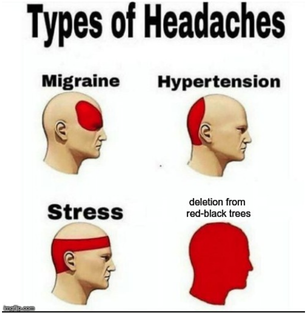
- long standing problem with a few proposed solutions [Kahrs 2001; Germane & Might 2014], but none of them totally satisfactory for us;
- complex invariant hard to describe precisely
- difficult to transcribe to our non-standard tree structure (bit-borrowing, leaf merging, etc.)

Take 1

We tried transcribing Kahrs' Haskell code directly to Coq without trying to fully understand it. But you guessed...

Take 1

We tried transcribing Kahrs' Haskell code directly to Coq without trying to fully understand it. But you guessed...



Using dependent types

Problem:

- not sure about how to do case analysis;
- not sure about the exact invariants;
- not sure about the auxiliary structures required.

Idea: use dependently-typed programming to guide programming process.

Auxiliary structures?

The intermediate data structures required for re-balancing the tree:

```
Inductive near_tree : nat -> nat -> nat ->
  color -> Type :=
| Bad : forall {s1 o1 s2 o2 s3 o3 d},
  tree s1 o1 d Black ->
  tree s2 o2 d Black ->
  tree s3 o3 d Black ->
  near_tree (s1 + s2 + s3) (o1 + o2 + o3) d Red
| Good: forall {s o d c} p,
  tree s o d c ->
  near_tree s o d p.
```

Re-balancing requires temporarily breaking the red-black tree invariants, hence the need for auxiliary structures.

Auxiliary structures?

The intermediate data structures required for re-balancing the tree:

```
Inductive near_tree : nat -> nat -> nat ->
  color -> Type :=
| Bad : forall {s1 o1 s2 o2 s3 o3 d},
  tree s1 o1 d Black ->
  tree s2 o2 d Black ->
  tree s3 o3 d Black ->
  near_tree (s1 + s2 + s3) (o1 + o2 + o3) d Red
| Good: forall {s o d c} p,
  tree s o d c ->
  near_tree s o d p.
```

Re-balancing requires temporarily breaking the red-black tree invariants, hence the need for auxiliary structures.

Take 2: Ltac

- use *tactics* to develop the program
- we ascribe strict types to each function, allowing to be completely sure that our code is correct
- as a side effect, we got a very clean specification
    - no “external” lemmas which can be easy to forget
    - all desired invariants were encoded into the types

Take 2: Ltac

- use *tactics* to develop the program
- we ascribe strict types to each function, allowing to be completely sure that our code is correct
- as a side effect, we got a very clean specification
  - no “external” lemmas which can be easy to forget
  - all desired invariants were encoded into the types

Programming Coq with tactics

The Pros

- “No brainer”: no need to fully understand the algorithm [Chlipala 2013]
- Easy to refactor: when underlying data structures change
- Quick fixes & adapting to changes

Programming Coq with tactics

The Cons

- You don't know what you're actually doing
- Readability: other people don't know what you're doing
- Semantics of Ltac changes frequently

Type-driven development?

"Type-driven" in what sense?

Regular development: design the algorithm, and then write types to check that you're correct.

Type-driven development: write types to declare what you want, and then code until it type checks

It type checks, ship it!

Type-driven development?

“Type-driven” in what sense?

Regular development: design the algorithm, and then write types to check that you’re correct.

Type-driven development: write types to declare what you want, and then code until it type checks

It type checks, ship it!

Type-driven development?

“Type-driven” in what sense?

Regular development: design the algorithm, and then write types to check that you’re correct.

Type-driven development: write types to declare what you want, and then code until it type checks

It type checks, ship it!

Type-driven development?

“Type-driven” in what sense?

Regular development: design the algorithm, and then write types to check that you’re correct.

Type-driven development: write types to declare what you want, and then code until it type checks

It type checks, ship it!

Applying the TDD methodology

At first, we had no clue about what the delete algorithm should look like!

We began with a complete specification:

```
Definition ddelete
  (d: nat)
  (c: color) (num ones : nat)
  (i : nat)
  (B : tree w num ones (incr_black d c) c) :
  { B' : tree (num - (i < num))
    (ones - (daccess B i)) d c |
    dflatten B' = delete (dflatten B) i }.
```

Applying the TDD methodology

At first, we had no clue about what the delete algorithm should look like!

We began with a complete specification:

```
Definition ddelete
  (d: nat)
  (c: color) (num ones : nat)
  (i : nat)
  (B : tree w num ones (incr_black d c) c) :
  { B' : tree (num - (i < num))
    (ones - (daccess B i)) d c |
    dflatten B' = delete (dflatten B) i }.
```

Applying the TDD methodology

Finding the missing auxiliary structure

We started to develop our function and found out that we needed to keep track of whether the height of a node has been decreased:

```
Inductive del_tree : nat -> nat -> nat -> color ->
  Type :=
| Stay : forall {num ones d c} pc,
  color_ok c (inv pc) -> tree w num ones d c ->
  del_tree num ones d pc
| Down : forall {num ones d},
  tree w num ones d Black ->
  del_tree num ones d.+1 Black.
```

Applying the TDD methodology

Refining the type

Now, we can write specifications for helper functions as well:

```
Definition balleft {lnum rnum lones rones d cl cr}
  (c : color)
  (l : del_tree lnum lones d cl)
  (r : tree w rnum rones d cr)
  (ok_l : color_ok c cl)
  (ok_r : color_ok c cr) :
{ B' : del_tree (lnum + rnum) (lones + rones)
         (incr_black d c) c |
         dflattend B' = dflattend l ++ dflatten r }.
```

Iterative development process similar to using holes and case-split iteratively in Agda or Idris.

Extraction

Two types of extraction:

- Extracting ML code from code defined using tactics
- “Extracting” a non-dependently-typed core of the algorithm within Coq (see ITP talk tomorrow)

Extraction

Two types of extraction:

- Extracting ML code from code defined using tactics
- "Extracting" a non-dependently-typed core of the algorithm within Coq (see ITP talk tomorrow)

Extraction

Two types of extraction:

- Extracting ML code from code defined using tactics
- “Extracting” a non-dependently-typed core of the algorithm within Coq (see ITP talk tomorrow)

Take 3: rewrite using `Program`

`Program` is a framework for dependently-typed programming in Coq [Sozeau 2006; 2008].

- Cleaner code: automatically generate type coercions for terms

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
    - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
    - `Error: the kernel does not support existential variables`
    - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Bad

- Many problems with unification engine: existential variables caused a lot of problems
- `program_simpl` was too aggressive sometimes, destroying goals in the process
  - Solution: disable `program_simpl`, unless the goal was directly solved by it.
- Bad error messages and mysterious failures
  - `Error: the kernel does not support existential variables`
  - Workaround: explicitly match on each argument that needs to be matched
- Performance issues with `Program`
- Simplifying and rewriting

Experiences with `Program`

The Good

- Readability and writability
- Obligation mechanism improves “modularity”
- Non-structural recursion using `measure`

Experiences with `Program`

The Good

- Readability and writability
- Obligation mechanism improves “modularity”
- Non-structural recursion using `measure`

Experiences with `Program`

The Good

- Readability and writability
- Obligation mechanism improves “modularity”
- Non-structural recursion using `measure`

Experiences with `Program`

The Good

- Readability and writability
- Obligation mechanism improves “modularity”
- Non-structural recursion using `measure`

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

The other alternative: `Equations`

Dependent pattern-matching compiler for Coq [Sozeau 2010; Sozeau & Mangin 2019].

- Even more readable code (Agda-like)
- `funelim` tactic supports easy pattern-matching in proofs
- No more Axiom K [Sozeau & Mangin 2019]

`Equations` was the perfect alternative for us, but currently some bugs prevent us from using it with MathComp.

e.g. issues #195, #212, #216, #217, #81 (closed), #179 (closed)

Our takeaways

- Dependent types in Coq are often useful. Use them as you see fit!
- Type-driven development is a great way to write programs that you don't know how to write!
- Tactics can be used to write programs, quite reliably
- Coq community needs to look at dependent types more (fix bugs, develop tools, etc.)

Our takeaways

- Dependent types in Coq are often useful. Use them as you see fit!
- Type-driven development is a great way to write programs that you don't know how to write!
- Tactics can be used to write programs, quite reliably
- Coq community needs to look at dependent types more (fix bugs, develop tools, etc.)

Our takeaways

- Dependent types in Coq are often useful. Use them as you see fit!
- Type-driven development is a great way to write programs that you don't know how to write!
- Tactics can be used to write programs, quite reliably
- Coq community needs to look at dependent types more (fix bugs, develop tools, etc.)

Our takeaways

- Dependent types in Coq are often useful. Use them as you see fit!
- Type-driven development is a great way to write programs that you don't know how to write!
- Tactics can be used to write programs, quite reliably
- Coq community needs to look at dependent types more (fix bugs, develop tools, etc.)

Our takeaways

- Dependent types in Coq are often useful. Use them as you see fit!
- Type-driven development is a great way to write programs that you don't know how to write!
- Tactics can be used to write programs, quite reliably
- Coq community needs to look at dependent types more (fix bugs, develop tools, etc.)

Future directions

- Editor support (esp. for `Equations`)
- Erasure of type indices à la [Brady, McBride & McKinna 2003]
- Showing only computationally-relevant terms
- Bug fixes and a better `program_simpl` tactic

Future directions

- Editor support (esp. for `Equations`)
- Erasure of type indices à la [Brady, McBride & McKinna 2003]
- Showing only computationally-relevant terms
- Bug fixes and a better `program_simpl` tactic

Future directions

- Editor support (esp. for `Equations`)
- Erasure of type indices à la [Brady, McBride & McKinna 2003]
- Showing only computationally-relevant terms
- Bug fixes and a better `program_simpl` tactic

Future directions

- Editor support (esp. for `Equations`)
- Erasure of type indices à la [Brady, McBride & McKinna 2003]
- Showing only computationally-relevant terms
- Bug fixes and a better `program_simpl` tactic

Future directions

- Editor support (esp. for `Equations`)
- Erasure of type indices à la [Brady, McBride & McKinna 2003]
- Showing only computationally-relevant terms
- Bug fixes and a better `program_simpl` tactic

Final remarks

Come to our ITP talk tomorrow at 16:30!

R. Affeldt, J. Garrigue, X. Qi, K. Tanaka. “Proving Tree Algorithms for Succinct Data Structures”.

https://github.com/affeldt-aist/succinct