Formalizing a Register Allocator in Coq

John Wiegley and Gregory Sullivan

BAE Systems, Burlington MA 01803, USA

Abstract. This experience report describes the process of formalizing a register allocation algorithm using the Coq proof assistant, and extraction to a Haskell implementation. We focus on how the Coq specification evolved from initial attempts—which seemed straightforward but later became unwieldy—to a final version that allowed for much easier proof. Our goal is to assist other teams who may be considering similar verification efforts.

Keywords: formal verification, register allocation, Coq, Haskell

1 Introduction

As part of implementing a Haskell-based cross compiler, from a C-like language named *Tempest* to a new hardware architecture called *SAFE*, we settled on the linear scan register allocation algorithm described by Wimmer and Mössenböck [2].

To gain confidence in our implementation (consistent with other formal verification efforts on the larger SAFE project[1]) we have formalized the algorithm in Coq [1]. Because the rest of our cross-compilation toolchain is implemented in Haskell, we extract Haskell code from the Coq development.

Our goal is to take the paper by Wimmer and Mössenböck as a specification (in English and pseudo-code) and formalize the algorithm in Coq so that *only correct implementations type-check*. After that, the only parameters of choice should be choosing which efficiency dimensions to optimize for.

The Coq definitions and proofs, and the generated Haskell code can be found at `https://github.com/jwiegley/linearscan`.

2 Design evolution

2.1 First design: Purely computational

In the first iteration, we approached the task much as any functional programmer might: data types carry computationally relevant information, functions are implemented in a straightforward fashion, and proofs are in terms of those types and functions. There are almost no dependent types involved, and very few inductive types. Most of the types used are records.

We quickly encountered two issues with this "purely computational" approach:

[1] See `http://www.crash-safe.org/` for more information on the SAFE project.

1. Proving termination of complex recursive functions is difficult.
2. Proving data structure invariants requires proving that functional updates maintain invariants. If we modify that structure, we must revisit many proofs.

The following function definition illustrates the above issues. `checkActiveIntervals` loops over the current set of intervals (an interval corresponds to the range of program locations where a single "virtual register" is live) and moves selected intervals into either the "handled" or "inactive" sets.

```
Coq < Definition checkActiveIntervals st pos : ScanStateDesc :=
Coq <   let fix go st st0 (ints : list (IntervalId st)) (pos : nat) :=
Coq <     match ints with
Coq <     | nil => st0
Coq <     | x :: xs =>
Coq <         let i := getInterval x in
Coq <         let st1 := if intervalEnd i < pos
Coq <                    then moveActiveToHandled x
Coq <                    else if pos < intervalStart i
Coq <                         then moveActiveToInactive x
Coq <                         else st0 in
Coq <         go st st1 xs pos
Coq <     end in
Coq <   go st st [seq fst i | i <- active st] pos.
```

Now consider the following two Lemmas, which claim that (1) `checkActiveIntervals` does not disrupt the next interval to handle; and (2) `checkActiveIntervals` does not introduce any new intervals.

```
Coq < Lemma checkActiveIntervals_spec1 : forall st st' pos,
Coq <   st' = checkActiveIntervals st pos
Coq <     -> nextInterval st = nextInterval st'.

Coq < Lemma checkActiveIntervals_spec2 : forall st st' i pos
Coq <   (H : st' = checkActiveIntervals st pos),
Coq <   ~ In i (all_state_lists st)
Coq <     -> ~ In (transportId (checkActiveIntervals_spec1 H) i)
Coq <             (all_state_lists st').
```

Because `checkActiveIntervals` operates by generating a new scan state on *each* iteration of the loop, we need to prove that all properties are maintained by each helper function.

As a result, we found ourselves having to break up functions further, and prove more auxiliary lemmas about each piece. After losing much time to this approach, we decided that evidence should be carried through each modification to prove the desired property. That is, rather than proving correctness "from the outside", the function was changed to produce the necessary evidence (as proof terms) needed by subsequent lemmas.

Lesson learned. Proving purely computational algorithms of this complexity meant using proof scripts to explore every possibility at each step of the algorithm: in effect, re-implementing the same algorithm in reverse, only now using tactics.

2.2 Second design: Proof-carrying data

In the second design of our algorithm, an attempt was made to reduce the complexity of proof scripts by adding evidence to the primary data structures, so that such evidence would be available wherever needed. Take for example an excerpt from an earlier version of the `ScanState` data structure, which records the overall state of several intermediary lists:

```
Coq < Record ScanStateV2 := {
Coq <     unhandled : list (nat * nat);
Coq <     active    : list nat;
Coq <     inactive  : list nat;
Coq <     handled   : list nat;
Coq <
Coq <     unhandled_sorted : StronglySorted (lebf (@snd _ _)) unhandled;
Coq <     lists_are_unique : uniq (map (@fst _ _) unhandled ++ active ++
Coq <                              inactive ++ handled)
Coq < }.
```

In this structure we have four working lists, and two properties about the lists. Property `unhandled_sorted` requires that the `unhandled` list is sorted by start position (2^{nd} element of each pair in list), and property `lists_are_unique` states that no interval index recurs across the four lists.

Lessons learned. While the idea of simultaneously maintaining data structures and invariants of those data structures is good, mixing data and propositions into one type does not quite work out, for the following reasons:

- It conflates data with predicates, which is generally a poor choice, because the number of predicates often grows over time.
- It forces the propositions to be considered whenever a value is mutated or created, even though it may be preferable to transform such values successively through different functions, and prove correctness of the composed mutation afterward.
  For example, if one were to pass around a sorted list structure, which combined a list with a proof of sortedness, it would disallow the possibility of unsorted intermediate state, which may still be corrected if the overall operation can prove that sortedness is maintained.

A guiding principle is to separate concerns as much as practical, which led directly to the third redesign, discussed in the next section.

2.3 Third design: Proof-Qualified Data (Regular Datatypes Qualified by Inductive Propositions)

After several iterations, where proving was found to be difficult and the resulting code looked quite unlike its non-dependent alternatives, we developed a methodology that is the primary message of this experience report. Note that this methodology is in no way novel or unknown, but we find it under-documented in the tutorial literature on Coq and so present it here to benefit others seeking to use Coq for verified programming tasks:

1. Design data-carrying types as inductive data types and records in `Set`, much like in a non-dependently typed language.
2. Constrain how such types may be constructed and mutated by encoding them as inductive propositions (in `Prop`). For example, the `StronglySorted` inductive predicate found in the Coq standard library.
3. High-level functions should take and produce data qualified by these propositions; lower-level worker functions may work on unqualified, intermediate values, so long as the composition is qualified.
4. Prove theorems about the properties required of the data, by induction on values qualified by the predicate(s).

Thus, the third iteration of our design divided each of the core data types into pairs:

- A non-dependent record, carrying computationally-relevant content only, in `Set`.
- Dependent, inductive types, carrying propositionally-relevant content only, in `Prop`.

The pairing of the two provides the "proof-carrying data" that was originally desired in the second design, but in a manner allowing for simple transformation of the underlying data when necessary, and induction on propositional evidence as required. The division of labor clarified not only the proofs, but made many constructions far simpler to manage. This is the final design that was settled upon, with one additional twist, to be continued in the next section.

2.4 Examples

As before, we define a data type *ScanStateDesc* that contains only data (no propositions). An excerpt is:

```
Record ScanStateDesc : Type := {
    unhandled : list (IntervalId * nat);   (* starts after pos *)
    active    : list IntervalId;           (* ranges over pos *)
    inactive  : list IntervalId;           (* falls in lifetime hole *)

    all_state_lists := map fst unhandledIds ++
                       active ++ inactive ++ handled
}.
```

Next, we define an inductive proposition named `ScanState`(of type Proposition, indexed by `ScanStateDesc`) that enumerates exactly the allowed methods for transitioning from one *ScanStateDesc* record satisfying `ScanState` to another *ScanStateDesc* record satisfying `ScanState`. Following are some excerpts:

```
Inductive ScanState : ScanStateDesc -> Prop :=
  | ScanState_nil :
    ScanState
      {| ... to fields to 0 or nil... |}
...
  | ScanState_moveActiveToHandled sd :
    ScanState sd -> forall `(H : x \in active sd),
    ScanState
      {| active           := rem x (active sd)
       ; handled          := x :: handled sd
         ... other fields unchanged ... |}
```

Now, if we want to prove a theorem about a `ScanState`-qualified `ScanStateDesc` record, we consider each case in the definition of `ScanState`.

3 Chains of Evidence Using Monads

During the course of development, it became clear that not only was evidence required of certain functions, but also a proof that this evidence maintains a special relationship with some initial condition. The primary case of this was proving well-foundedness of the main algorithm.

Well-foundedness may be demonstrated in Coq 8.4 by establishing a mapping from some input argument to a natural number, and then proving that this value only decreases with each recursive call. If we further view an algorithm as a composition of smaller functions, then each member of the composition must either preserve this value, or cause it to decrease at least once.

In order to prove that order is maintained, and that we are ultimately "productive" (i.e., a decrease in the initial value, the requirement of well-foundedness), we encode this requirement in the type of each composed function. That is, the type must show that the function's input is $\leq$ the initial value, and that the output is $\leq$ or $<$ the initial value. If at least one output is $<$, and each subsequently composed function is only $\leq$, we have demonstrated well-foundedness overall.

Such specialized composition—where a context is preserved across function calls—is a notion captured exactly by monads from category theory. What we are describing above is specifically an *indexed state monad*, where a certain state (a relation with our initial value) is maintained throughout the composition, and we may specify for each function what its input and output requirements are of this relation.

Here is the sort of function type we end up with:

```
Coq < Definition handleInterval {pre : ScanStateDesc}
Coq <   : SState pre SSMorphHasLen SSMorphSt (option PhysReg).
```

This type says: this function accepts some incoming state which is related to an original state `pre` in that it still has work to done (`SSMorphHasLen`); its output state reduces the scope of this work (`SSMorphSt`). The result value is a physical register, if one was able to be allocated. Note that this signature tells us that calling this function provides sufficient evidence to prove termination (due to the `SSMorphSt` relation), as will any composition including this function.

Working with a monad like this permits the management of evidentiary state to happen "behind the scenes", clarifying the code by removing explicit proof management. In the version of our implementation before this change, proving these relationships became quite time-consuming, and obscured the real work being done.

By this means, the final well-foundedness proof of the main algorithm became a one-liner, while at the same time each component function of the algorithm was greatly simplified, reading similarly to an equivalent Haskell implementation.

4 Code extraction

A primary goal of this endeavor, from the very beginning, was extraction to Haskell. The question was whether Coq and formal methods would be up to the task of building a Haskell library in a better way.

About midway through the development, focus was turned toward producing reasonable Haskell code through Coq's built-in extraction facilities. These allow for:

- Extraction of functions and types (those not in Prop), and their related definitions.
- Association of some types and constructors with native Haskell types.
- Renaming some functions to use their direct Haskell counterparts, requiring that one implicitly trust these to be both total and correct.

There is unfortunately no published meta-theory for the extraction process from Coq to Haskell, and indeed some bugs were encountered during the course of this project (including one where type variables were inexplicably swapped in a constructor whenever auto-inlining was used).

Even still, the ability to write code in Coq and have it so easily converted to naive Haskell was incredibly useful, and an area that we hope sees further development. The resulting Haskell code was quite simple and straightforward, so that it is amenable to optimization by the GHC compiler.

References

1. The Coq Development Team: The Coq Reference Manual, version 8.4 (Aug 2012), available electronically at `http://coq.inria.fr/doc`
2. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. pp. 132–141. VEE '05, ACM, New York, NY, USA (2005), `http://doi.acm.org/10.1145/1064979.1064998`