Experience Report: Implementing a register allocator for Haskell in Coq

John Wiegley and Gregory Sullivan

BAE Systems, Burlington MA 01803, USA

Abstract. We describe the process of formalizing a register allocation algorithm, and its correctness properties, in the Coq proof assistent, and subsequent extraction of a Haskell implementation. Rather than simply state the final specification in Coq, as if we had constructed it in one pass, we focus on the process of evolving the Coq specification from our initial attempts—which seemed straightforward but later became unwieldy—to a final version that allowed for much easier proof. Our goal in this experience report is to assist other teams who may be considering similar algorithm verification efforts, in the hope that those teams can learn from our mistakes.

Keywords: formal verification, register allocation, Coq, Haskell

1 Introduction

As part of implementing a 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 [**?**].

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 [**?**]. 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 typecheck*. After that, the only parameters of choice are choosing which efficiency dimensions to optimize for.

In Section 5, we give a brief overview of the linear scan register allocation algorithm and its datatypes, focusing on correctness properties. In Section 2, we describe the design iterations as we discovered strategies for applying Coq to this sort of combined verification and implementation task. We conclude with lessons learned, in the hope that our experience will help others embrace the methodology of formal development, proof, and code extraction (while avoiding some of the pitfalls).

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

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. Proving termination of loops is difficult.
2. Proving invariants over data structures requires proving that every function maintains all invariants. When we want to modify a function's logic, we need to 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))
Coq <              (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 (active st) pos.
```

First, proving termination of any function defined as a fixpoint is difficult in Coq; in particular, it is difficult to use induction, which is a pervasive proof technique in Coq.

Next, 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 the `checkActiveIntervals` function 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 and more auxiliary lemmas about each piece. After losing much time to this approach, we decided that evidence should be carried through each modification step to prove the desired property. That is, rather proving correctness "from the outside", the function itself was changed to manage the evidence, which removed the need for helper 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 OldScanState := {
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 record type to dwell in `Type` rather than `Set`, and so its inhabitants cannot be used by functions or data structures restricted to `Set`.
- Any time such values must be mutated, the correctness of the mutation must be separately proven *at each instance where it occurs*. That is, the properties cannot be proven generally, but must be constantly re-proven in each new context.
- 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 manitained.

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 unqualifed, 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).
5. (minor point) If the only role of a proposition is to indicate that a property is held or not (for instance, that a list has a certain length), it is better to do

so using a boolean-returning function called from an existential data type that pairs the qualified value with the function's result: an example would be `{ l : list a | size l == 4 }`, or specialized by a data constructor as:

```
Coq < Inductive listn {a} := mklistn l n of (@size a l == n).
```

This enables use of small-scale reflection (see the ssreflect library), which has benefits beyond the scope of this paper. We mention it here only to underscore that we do not recommend all propositions be made inductive, but only those more information rich than simple yes/no properties.

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 original 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 datatype *ScanStateDesc* that contains only data (no propositions). An excerpt is:

```
Record ScanStateDesc : Type := {
    nextInterval : nat;
    IntervalId   := fin nextInterval;

    unhandled : list (IntervalId * nat);   (* starts after pos *)
    active    : list IntervalId;           (* ranges over pos *)
    inactive  : list IntervalId;           (* falls in lifetime hole *)

    unhandledIds    := map fst unhandled;
    unhandledStarts := map snd unhandled;
...
    all_state_lists := 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
      {| nextInterval     := 0
       ; unhandled        := nil
       ; active           := nil
       ; inactive         := nil
       ; handled          := nil
       ; intervals        := V.nil _
       ; assignments      := V.nil _
       ; fixedIntervals   := V.const None _
       |}
...
  | ScanState_moveActiveToHandled sd :
    ScanState sd -> forall `(H : x \in active sd),
    ScanState
      {| nextInterval     := nextInterval sd
       ; unhandled        := unhandled sd
       ; active           := rem x (active sd)
       ; inactive         := inactive sd
       ; handled          := x :: handled sd
       ; intervals        := intervals sd
       ; assignments      := assignments sd
       ; fixedIntervals   := fixedIntervals sd
       |}

...
```

Now, if we want to prove a theorem about a `ScanState`-qualified `ScanStateDesc` record, we consider each case in the definition of `ScanState`. For example, to prove that the unhandled interval list is sorted, the proof is:

```
Theorem unhandled_sorted `(st : ScanState sd) :
  StronglySorted (lebf snd) (unhandled sd).
Proof.
  ScanState_cases (induction st) Case; simpl in *.
  - Case "ScanState_nil". constructor.
  - Case "ScanState_newUnhandled".
    rewrite /unh.
    by apply/StronglySorted_insert_spec/StronglySorted_widen/IHst.
  - Case "ScanState_moveUnhandledToActive". inv IHst.
  - Case "ScanState_moveActiveToInactive". apply IHst.
  - Case "ScanState_moveActiveToHandled". apply IHst.
  - Case "ScanState_moveInactiveToActive". apply IHst.
  - Case "ScanState_moveInactiveToHandled".  apply IHst.
Qed.
```

3 Proof morphisms

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}
Coq <   : SState pre SSMorphHasLen SSMorphSt (option PhysReg).
```

This function signature tells us that with regard to some original state `pre`, we require that the state upon entering this function is related to this original state such that there is work yet to be done (i.e., we have proven productivity), and that the output state demonstrates productivity. Use of this function alone gives sufficient evidence to prove termination, as will any composition that includes 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.

```
Proof.
  intros; clear; by case: ssinfo' => ?; case=> /= _ /ltP.
Qed.
```

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 implicit 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 glaring 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 result Haskell code was quite simple and straightforward, which should lend fairly well to optimization by the GHC compiler.

5 Register Allocation Algorithm and Correctness Properties

The algorithm we implemented is fully documented in the original paper[**?**]. We extend that documentation by presenting several requirements that were implicit in their presentation, but became necessarily explicit in the Coq formalization. Each requirement stated below was encoded in either a type or a lemma, and proven during the course of development.

5.1 Use positions

When code is linearized for the purpose of register allocation, each position where a variable is used is termed a *use position*. There is also a boolean flag to indicate whether a register is required at this use position or not.

Requirements

– All use positions must be given odd addresses, so that ranges do not extend to the next use position.

5.2 Ranges

A list of use positions constitutes a range, from the beginning use position, to one beyond the last. A range may also be extended so that its beginning and end fall outside its list of use positions. This is done in order to provide explicit coverage of loop bodies, to avoid reloading registers each time the loop body is executed.

Requirements

– The list of use positions is ordered with respect to location.
– The list of use positions is unique with respect to location (no recurring use positions).
– The beginning of a range is $\leq$ its first use position location.
– The end of a range is $>$ its last use position location.

5.3 Intervals

An ordered list of ranges constitutes an *interval.* In the majority of scenarios, intervals are equivalent to ranges, since only a single range is used. However, adding the concept of intervals allows for *lifetime holes* between component ranges, that are otherwise allocated to the same register.

Requirements

– The list of ranges is ordered by the first use position in each range.
– The list of ranges is non-overlapping.
– The beginning of the interval corresponds to the beginning of its first range.
– The end of the interval corresponds to the end of its last range.

5.4 Scan state

During the course of the algorithm, a position counter is iterated from the first code position to the last. Six separate details are managed using a collection of lists:

1. A list of yet to be allocated intervals (unhandled intervals).
2. A list of active intervals (those covering the current position, and presently allocated).
3. A list of inactive intervals (those having a lifetime hole covering the current position).
4. A list of handled intervals.
5. A mapping from registers to active/inactive/handled intervals.
6. A mapping from intervals (those not unhandled) to registers.

Requirements

- The list of unhandled intervals is ordered by start positions.
- The first 4 lists, taken as a set, must contain no recurring intervals, and have no overlapping intervals.
- All register indices must be $<$ the maximum register index.
- The number of intervals in the active list be $<$ the maximum register index.