Proof Pearl: Program-ming reachability algorithms in Coq

James McKinna, Dan Synek, and Eelis van der Weegen

Institute for Computing and Information Sciences
Radboud University Nijmegen
Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

Abstract. We present a comparison of approaches to the representation and verification of non-structurally recursive algorithms in the type theory CIC of the Coq proof assistant. We illustrate our ideas in the context of reachability algorithms on (finite) graphs.
Our approach makes use of Sozeau's Program machinery, a relatively recent addition to Coq, which permits a very clean representation of functional programs in CIC. We consider: a version of the algorithm which only focuses on the termination argument, for which we then give a direct proof of partial correctness; a definition and proof using Program 'in one go'; and a final variant, where we confine the use of Program to total correctness, proving partial correctness via the graph of the function.
This leads to some observations for Coq users and implementors concerning the traditional separation of concerns in program verification.

1 Introduction

This paper makes another contribution to the literature and culture of systematic formal reasoning in constructive type theory about *non-structurally* recursive functions. We consider algorithms for the problem of reachability in finite directed graphs, represented in the type theory CIC of the proof assistant Coq.

1.1 The Problem

What does the application-oriented developer of proofs do to streamline the task of checking or proving properties of algorithms, when these do not enjoy 'natural' representation in the (programming) language of their chosen proof assistant?

A recurring idea when proving programs correct is the separation of concerns. We first want to define a program, then prove its partial correctness and then prove that it terminates for all suitable arguments. When working in type theory, this is not a problem when the function can be defined by *structural* recursion, since termination is then a consequence of its definition. Definition by *non-structural* recursion is more problematic, as we then cannot express it as a function until we have proved it terminates. We typically might do this by well-founded recursion, but that can lead to a term whose intensional computational behaviour is at odds with the recursive specification we have in mind.

Another problem, also a separation of concerns, is how to choose when to specify the properties of the function we are defining. Using the propositions-as-types paradigm we can specify the properties by an expressive Σ-type and then reason about the function from the input-output properties so specified. But this means that we need to anticipate the properties we need the function to have all at once by giving it an expressive enough type. An alternative is to prove the properties of the function as needed; but this means that we over and over again need to do some kind of well-founded induction which is appropriately correlated with the well-founded recursion defining the function; this seems unnecessarily complicated, and can certainly become so in practice.

A canonical solution, and the one we elaborate upon here, is to define an induction principle once and for all from the definition of the function. One way to do this is to inductively define a relation, the extensional graph of the function, corresponding to the recursive call structure of the function we are defining. The induction principle derived from this relation is precisely the one needed to prove any property about the function, since its definition captures exactly the intermediate recursive calls arising from a given call to the function, and hence the corresponding induction hypotheses.

Having separated the induction principle from the function we wish to define we can prove the properties we want using the induction principle induced by the graph *without reference to the well-definedness or otherwise of the function.* We can then define the function using the `Program` command using the minimal criteria to establish termination, namely inhabitation of the graph. Indeed, being inductively defined, the graph encapsulates the *minimal* properties which any call of the function, assumed to be well-defined, must satisfy; thus we may recover (a version of) least fixed-point semantics in the context of a total type theory.

This paper shows a way to factor that development, in the case of a reachability algorithm, into a termination argument (justifying the existence of a function) and a separate partial correctness proof, which makes no commitment to the existence of the function. The key enabling device has already been alluded to: a mechanism for obtaining the inductive hypotheses asserting the well-definedness of intermediate recursive calls.

1.2 Related Work

Unsurprisingly, there is a great deal of related work; this touches on some of the oldest problems and techniques in computer science, as well as the most recent. Our original inspiration for considering the inductively-defined graph associated with a function comes from McCarthy's seminal work on *computational* (or *recursion*) *induction*[1]. Gérard Huet was another early influence, describing inductively-defined families in type theory as “a kind of typed Prolog”. In unpublished work[2] the first author (with Conor McBride) has explored its application in the context of EPIGRAM function definitions; the graph is an example of their version of Wadler's idea of *view* [3].

Sozeau's recent PhD thesis work [4, 5] introduced the impressive `Program` suite of commands and tactical support to the Coq system. We will explain its behaviour and usage below.

Slind's PhD thesis work [6] offers users of both HOL and Isabelle/HOL a great deal of similar functionality with the TFP package, but a direct comparison between the approaches seems difficult since CIC is by design a *computational* meta-theory, distinguishing *intensional* objects (algorithms/functions) whose evaluation is part of the power of the theory, from their logical (extensional) behaviour established by proof.

Working at Sophia-Antipolis with the Coq system itself, Bertot and Balaa [7] considered the problems associated with functions defined by well-founded recursion. Their work draws attention to many of the problems arising from attempting to relate intensional behaviour of such functions to their extensional properties given by their fixed-point equation. Later, Barthe and his co-workers [8, 9] developed the `Function` machinery for generating the inductively-defined graph and its associated induction principle automatically from a CIC-definable function. Their work furthermore permitted a clean syntax for function definition both by structural recursion and by well-founded recursion. It is unfortunate that the current implementation of these ideas remains buggy, and does not integrate well with the new `Program` infrastructure.

A companion approach to reasoning with graphs is the so-called 'Bove-Capretta' method of working with inductively-defined *domain predicates*[10–12]. As the name implies, these predicates define the domain of definition of a function; they are used as a structure over which to do recursion in order to define otherwise non-structurally recursive functions in type theory. Extensionally, they may be seen as precisely the domains of the corresponding inductively-defined graphs. Space forbids a detailed comparison between their approach and ours.

Work on algorithms on directed graphs (we crave the indulgence of the reader over the clash in these two uses of the term 'graph') and their correctness proofs [13, notably] goes back at least as far as McCarthy's work, and a survey seems an impossible endeavour here. The classical literature presents such things in imperative, not functional, style. As regards *machine-checked* developments in proof assistants, as much to our surprise as were Moore and Zhang's earlier observations about Dijkstra's SSSP algorithm [14], there seem comparatively few examples of how to do this kind of thing without relatively heavy machinery. A notable comparison point is Hurd's formalisation in HOL in the context of a (much larger) development [15]. Tobias Nipkow drew our attention to a very closely related, but entirely independent, development by Nishihara and Minamide in Isabelle/HOL [16]; their approach involves a *nested* recursive definition of depth-first search, with the associated complications in reasoning.

1.3 Contribution

In the context of hybrid systems verification in the the Coq system, we needed to check properties of various graph reachability algorithms. We isolated the work of this paper from that much larger development.

Our principal contribution is to compare various versions of how to define, and prove the correctness of, the algorithm `reachables` explored in detail below.

prototype We recap the imperative iterative specification of the algorithm defined in pseudocode, together with a sketch of its correctness proof.

simplest version We first define a 'plain' version of the algorithm, as a tail-recursive 'worker' function `reachables_worker`, and a flat 'wrapper' function `reachables`. These operate on lists (representing subsets), so we need to introduce non-duplication hypotheses for the representation to be valid. We also define the 'step' function `rstep`, implementing the loop body in the imperative prototype. The only non-trivial proof obligation in the use of `Program` is that required to prove termination. This requires formalisation of the termination invariant and the measure function.

version 0 Now, having successfully defined these functions, in order to prove anything about them, one approach is to develop directly an induction principle 'by hand' for `reachables_worker` and `reachables`, which isolates their logical properties from the particular method (by well-founded recursion) which `Program` uses internally to build definitions. In fact, by a quirk of the `Program` implementation, we need to tweak the functions slightly, yielding `reachables_worker0` and `reachables0`. The induction principle we derive is moreover 'polluted' with the need to explicitly project lists out of the Σ-types used in our definitions. We use the induction principle to prove soundness and completeness of our definitions; we refer back to a collection of lemmas proved about the various predicates and invariants, especially regarding how they interact with the step function `rstep`.

version 1 We next consider a direct proof of correctness, organised as a single `Program Fixpoint` definition, of `reachables_worker1` and `reachables1`. This version yields a partial correctness and termination argument most closely resembling the classical Dijkstra-Hoare point of view. Both the input predicates and result types become entangled with information necessary only for termination. Moreover, we need to use the hypotheses arising from well-founded induction in order to reason inductively as if the function is well-defined.

version 2 We then factorise the correctness proof, by introducing the relational specification of the inductively-defined graph of `reachables_worker` (we ignore the case of `reachables`, as it has no interesting inductive structure). By induction on this relational specification we can prove the desired (partial) correctness properties; the termination argument is as before, but now the initial conditions are as in the first version, while the result type specifies that the function witnesses the graph relation. The proof of this is very nearly automatic, and could be made so, we believe, in a future version of `Program`.

version ... All of the foregoing is, in fact abstract with respect to the properties of the step function `rstep`. Indeed, we could keep abstracting, based on the graph relation idea, moreover in a compositional way. But for the sake of this paper (and the reader!) we leave such extensions to future papers.

1.4 Outline

We begin in Section 2 with the classical imperative specification and verification of an abstract reachability algorithm. We then show in Section 2.1 how this gives rise to a proto-definition in the Gallina specification language of Coq, remarking on the limitations of the raw typechecker in attempting to check wellformedness (in particular, termination) of such definitions.

We then sketch in Section 2.2 how Sozeau's `Program` machinery can help us, and in particular how it might be used to support the separation of concerns.

We then describe the various versions of our concrete (depth-first) functional implementation of reachability, and show how the use of `Program` leads to a less-or-more smooth organisation of the verifications which arise in each case. We discuss the consequences each version has for the separation of concerns and the pragmatics of working with Coq.

Finally, we conclude with some observations about the further abstraction inherent in our development and the prospects for extensions and future work.

2 Specifying and verifying the algorithm

The graph reachability algorithm we formalise can be specified imperatively. Its definition and correctness proof are routine, and included only for the sake of completeness.

pseudocode prototype We assume given a finite directed graph G. The operators `+` and `-` denote set union and set difference, respectively, defined on subsets of the set `vertices` of the vertices in G.

```
Algorithm reachables.

inputs: subset "start" of vertices.
pre-condition:

output: subset "result" of vertices.
post-condition:
  for all v, (v in result) <-> (v reachable from start).

begin
locals: subsets "visited" and "waiting".
  visited:= {};
  waiting:= start;
  while (waiting <> {})
   {
    pick w in waiting;
    visited:= visited + {w};
    waiting:= (neighbours w) + waiting - visited;
   };
  result:= visited;
end
```

informal proof The partial correctness of algorithm `reachables` follows from the following equivalent of the post-condition:

```
post-condition: for all v,
  (v in (waiting + visited)) <-> (v reachable from start).
```

This is because on termination of the loop we have `waiting = {}` and hence on termination of the algorithm, `result = waiting + visited`. The condition trivially holds on loop entry, since then `waiting + visited = start`. Unfortunately, the condition alone is *not* a loop invariant.

The subset of vertices `R` reachable from `start` may be characterised inductively as the *smallest* subset `S` satisfying two properties:

- `for all v, v in start -> v in S`
- `S is closed under neighbours`

To show `R` contains `result` on termination (*soundness*) amounts to observing that at each iteration, the subset `waiting + visited` grows only by the addition of neighbours, and hence, inductively, by vertices reachable from `start`.

Similarly `waiting + visited` always satisfies the first property, that is to say `for all v, v in start -> v in (waiting + visited)` is a loop invariant. Hence on termination of the algorithm, `result` will contain `R` (*completeness*) provided we can satisfy the second property.

So we are left with Dijkstra's invariant which establishes this:

```
invariant: for all v,
  (v in (neighbours(visited))) -> (v in (waiting + visited)).
```

On exit, similarly to above, the property reduces to

```
for all v, (v in (neighbours(result))) -> (v in result).
```

and hence `result is closed under neighbours` as required. It trivially holds on loop entry, where `neighbours(visited) = neighbours({}) = {}`. Finally, one proves that it is maintained on each iteration by inspection of the loop body.

Total correctness follows if we can show that a variant measure decreases on each iteration: the size of `vertices - visited` is a suitable such measure, *provided* we can show that on each iteration, a 'new' vertex is added to `visited`. We remark without further comment that this can be secured with the additional loop invariant stating that `waiting` and `visited` are disjoint.

2.1 Towards defining the algorithm in Coq

We begin with an initial preamble loading list utilities and other library files (here omitted; please see the full development for details), followed by an algebraic structure for directed graphs, and reachability, characterised in the usual way as the reflexive-transitive closure of a binary relation.

```
Variables (State: Type) (trans: State -> State -> Prop).


Inductive reachable: State -> State -> Prop :=
  | reachable_refl s: reachable s s
  | reachable_next a b c: reachable a b ->
      trans b c -> reachable a c.

Record DiGraph: Type := Build
  { Vertex: Set
  ; Vertex_eq_dec: forall (v v': Vertex), decision (v = v')
  ; vertices: list Vertex
  ; vertices_exhaustive: forall v, In v vertices
  ; edges: Vertex -> list Vertex
  ; edges_NoDup: forall v, NoDup (edges v)
  }.
```

We then define reachability in the graph from an initial set of vertices `start`; this property is rather straightforwardly closed under the edge relation.

```
Variable G: DiGraph.

Let Edge (v w: Vertex G): Prop := In w (edges v).

Let SubsetV := list (Vertex G).
Let emptyV : SubsetV := []. Hint Unfold emptyV.
Let addV v vs : SubsetV := v :: vs. Hint Unfold addV.

Variable start: SubsetV.

Hypothesis NoDup_start: NoDup start.

Let reachable v: Prop := exists s,
  In s start /\ reachability.reachable Edge s v.

Lemma reachable_start v: In v start -> reachable v.
```

As a pure functional program (CIC term), we factor the imperative control structure into a *worker* which expresses the result as a (tail-recursive) function of the local variables, and a *wrapper*, which initialises them.

pseudo-Coq fragment The wrapper function (minus typing decoration) is very straightforward.

```
Definition reachables
  := reachables_worker emptyV start.
```

The functional prototype of the worker function relies on a definition of a function *`rstep`* implementing the assignments of the loop body. The *`pick`* operation is modelled by pattern matching on the *`waiting`* list: when the list is empty, the function terminates, returning a *`result`* list given definitionally by (the final value of) *`visited`*.

```
Fixpoint reachables_worker (visited waiting: SubsetV) :=
  match waiting with
  | nil => visited
  | w :: ws => reachables_worker (addV w visited) (rstep visited w ws)
  end.
```

where the step function `rstep` may be given directly in Coq as follows:

```
Definition rstep vs w ws :=
(subtrV (edges w) ((addV w vs) ++ ws)) ++ ws.
```

Since the worker's recursion is non-structural, we cannot give it as an ordinary `Fixpoint`/`match` style definition in Coq's Gallina vernacular syntax. The `match` construction makes 'ML-style' pattern matching available to the user (the real story is much more powerful, but correspondingly more complicated, thanks to the dependent types in CIC). The `Fixpoint` construction only supports the definition of structurally recursive functions, whose termination checking is handled by a hard-wired syntactic check ('guarded by destructors') in the Coq typechecker. This makes prototyping the program, and its correctness proofs, fall at the first hurdle, and has been seen as a long-standing disadvantage of working in type theory for certified programming.

2.2 What does Program do?

Sozeau's `Program` extensions to this language give the programmer a much more flexible range of action:

- the syntax is upwardly compatible (one writes `Program Fixpoint` etc.);
- termination may also be specified by a measure function into a well-founded ordering;
- the input and output types of the function may be constrained by predicates in a way which is transparent to the function body.

Rather than perform a yes/no syntactic check, the `Program` machinery then generates proof obligations for the user, much in the style of PVS predicate subtyping, induced by the structure of the program one is attempting to define.

The 'definition' is not accepted by the typechecker until each pending obligation has been discharged: so in this sense, the typechecker cannot accept definitions which have not been proved terminating. However, the algorithm for typechecking-modulo-predicate-subtyping which drives this machinery does allow the programmer:

- to write partial functions, if a suitable conservative domain predicate may be specified; this leads to proof obligations on the application of such functions;
- to constrain the result type, which will lead to an obligation to prove that property of the computed answer;
- to delay the obligation to show termination via the ordering.

Further, the typechecking algorithm follows the Fixpoint/match structure given by the programmer, so match will typically lead to proof obligations corresponding to each case, and use of Fixpoint generates *an inductive hypothesis* (guaranteed by the well-founded recursion driving the termination proof) providing, for suitably constrained inputs, *the existence of recursive calls* of the function.

Now, Coq users may specify functions, their termination measures, and their input-output behaviour in advance of, or rather hand-in-hand with, ensuring the well-definedness of such objects. However, in doing so, it is easy to lose sight of the classical separation of concerns between termination and correctness: while these are given to the user of Program as separate obligations to discharge, nevertheless the 'script' object which defines such a function must be completed before any other useful work may be done.

It seems that one must still prove total and partial correctness together. Not only that, but an object successfully defined by Program is only *extensionally* equivalent to the programmer's recursive specification: necessarily so, as it is given internally by appeal to recursors over well-founded orderings. To prove anything about such a function after defining it leaves the user with the uncomfortable task of exhuming from Program's internals exactly those appeals to well-founded recursion/induction needed to massage the definition into the right form.

3 Using Program to define and verify the algorithm

3.1 Simplest Version

The worker function itself may be given in the extended syntax as follows.

```
Program Fixpoint reachables_worker (visited: SubsetV)
  (waiting: { ws | Termination visited ws })
  {measure measureV visited}: SubsetV :=
  match waiting with
  | nil => visited
  | w :: ws => reachables_worker (addV w visited) (rstep visited w ws)
  end.
```

The waiting argument is given a Σ-type, whose predicate Termination expresses the invariant required to show decrease of the measure described earlier.

```
Definition Termination (vs ws: SubsetV): Prop :=
  NoDup ws /\ Disjoint ws vs.

Definition measureV (vs: SubsetV): nat :=
  length (subtrV (vertices G) vs).
```

This gives rise to the first of two generated obligations:

```
  NoDup (w :: ws) /\ Disjoint (w :: ws) vs ->
    measureV (w :: vs) < measureV vs.
```

and without building in the termination invariant `Termination` this lemma would not be provable. Additionally we see a second obligation generated, expressing that the invariant holds for the arguments passed in the recursive calls. In other words, it must be shown that the invariant is preserved by `rstep`:

```
  Lemma Termination_preserved vs w ws:
    Termination vs (addV w ws) ->
    Termination (addV w vs) (rstep vs w ws).
```

The proof of this obligation is similarly straightforward. After both have been discharged, the constant `reachables_worker` is actually defined by `Program`. It is defined in terms of `Fix_measure_sub`, which encapsulates the machinery used for induction over a measure, using the accessibility predicate `Acc`.

The non-recursive wrapper is now definable, generating *no* proof obligations.

```
Program Definition reachables: SubsetV
  := @reachables_worker emptyV start.
```

Correctness Now that we have a working definition, it is time to start worrying about correctness. For this algorithm, correctness can be split into soundness and completeness. More specifically, we would like to establish the following Σ-type for the `reachables` function:

```
Program Definition reachables: { rs | Specification rs }
:= @reachables_worker emptyV start.
```

where `Specification` consists of two conjuncts:

```
Definition Specification rs : Prop := Sound rs /\ Complete rs.
```

Soundness specifies that everything computed is indeed reachable:

```
Definition Sound (ss : SubsetV): Prop := forall v, In v ss -> reachable v.
```

Completeness specifies that everything reachable is computed. We inductively generalise this notion, and establish its relationship to the loop exit property:

```
Definition Complete (ss: SubsetV): Prop :=
  forall v, reachable v -> In v ss.
Definition GComplete (vs ws rs: SubsetV): Prop :=

  closed_under Edge rs /\ incl ws rs /\ incl vs rs.

(* termination lemma: when we finish, we have what we want *)
  Lemma gcomplete_complete vs: GComplete [] start vs -> Complete vs.
```

3.2 Version 0: Correctness by separate induction

In a grossly simplified view of the problem, one typically proves properties of a recursively defined function by induction on the argument on which the algorithm structurally recurses. This way, in the case of a recursive call, the function invocation naturally unfolds to some code applied to a simpler application, which is (hopefully) precisely the subject of the induction hypothesis.

For functions defined by an ordinary `Fixpoint`, this strategy works "out of the box", without any additional machinery. However, the function `reachables_worker` we defined is not an ordinary `Fixpoint` definition. As described above, it is expressed indirectly in terms of `Fix_measure_sub` which does some intricate recursion on (proofs of) the accessibility predicate `Acc`, using a constant corresponding to the measure-decreasing obligation we proved above. But that's not what the user thinks of as the semantics of the recursive definition. Consequently, it is most unnatural to try to appeal to the *actual* decreasing argument's induction principle (`Wf_rec` or something), since from a user's perspective, the argument to the recursive call is what gets "smaller". Or rather: such a call is "earlier" in the course of values computed on the way to the top-level call. So what we really want is an induction principle saying:

```
Lemma reachables_worker0_ind (P: forall (vs ws rs: SubsetV), Prop)
  (Pbase: forall vs, P vs emptyV vs)
  (Prec: forall vs w ws rs, P (addV w vs) (rstep vs w ws) rs ->
    P vs (w :: ws) rs):
  forall vs ws, P vs (‘ws) (reachables_worker0 vs ws).
```

We can prove such an induction principle by unfolding the `Wf_measure_sub` machinery and following the recursion over `Acc`. In fact, however, we do not find ourselves in this ideal world, at least not at first. We derive automatically (but with some pain) the following principle:

```
Lemma reachables_worker0_ind_aux
 (P: forall (vs: SubsetV) (ws: {l | Termination vs l}), SubsetV -> Prop)
 (Pbase: forall vs a, P vs (exist _ emptyV a) vs)
 (Prec: forall vs w ws p rs,
   P (w :: vs) (exist _ (rstep vs w ws) p) rs -> forall mp,
   P vs (exist _ (w::ws) mp) rs):
  forall visited waiting, P visited waiting (@reachables_worker0 visited waiting).
```

from which we may, with further work, get the desired induction principle. One sees in `reachables_worker0_ind_aux` the extra 'junk' in the form of existential witnesses such as `(exist _ (rstep vs w ws) p)` for termination which are only there because required by `Program`. In an ideal world, we would be able to prove properties like soundness and completeness while oblivious to termination concerns (and indeed, in section 3.4 below we will show a way to complete this separation of concerns).

Nevertheless, using the derived induction principle, we can prove soundness and completeness:

```
  Lemma sound vs ws: Sound vs ->
    Sound (`ws) -> Sound (reachables_worker0 vs ws).

  Lemma complete vs ws: Invariant vs (`ws) ->
    GComplete vs (`ws) (reachables_worker0 vs ws).
```

But notice now the wrinkle in the ointment: the lemmas speak about the values (`` `ws ``) which are *first projections* from the decorated Σ-types, and in practice this kind of niggling detail, and its proliferation into other goals, can overwhelm the non-expert user.

3.3 Version 1: Integrated correctness

A very palatable alternative to the last solution involving a custom induction principle, and in harmony with forty or fifty years of thought in imperative program verification, is to integrate the correctness statement into the original definition. That is, instead of writing a `Program Fixpoint` which produces a bare list, we have it produce a Σ-decorated list which bundles up the soundness and completeness proofs. Just as with the classical informal sketch of the proof, this forces us to add new invariants to the input argument types in order to make the resulting extra proof obligation provable. The result looks like this:

```
Program Fixpoint reachables_worker1
  (visited: { vs | Sound vs })
  (waiting: { ws | Sound ws /\ Termination visited ws /\ Invariant visited ws})
    {measure measureV visited}:
      { rs | Sound rs /\ GComplete visited waiting rs } :=
  match waiting with
  | nil => visited
  | w :: ws => reachables_worker1 (addV w visited) (rstep visited w ws)
  end.
```

We now get the all-too-imaginable proof obligations, for example in the case of an empty waiting list:

```
Next Obligation. (* the result in the nil case meets the spec *)
  (* Sound visited /\ GComplete visited [] visited *)
```

In all, there are five outstanding proof obligations, namely to check:

- that the exit condition of the recursion does indeed yield a sound and complete set of vertices;
- that the argument (`addV w visited`) to the recursive call is indeed sound;
- that the conjunction of the soundness, termination and Dijkstra invariants is indeed preserved by `rstep`;
- that the termination measure does indeed decrease;
- that the Dijkstra invariant implies generalised completeness.

It is good to know that none of them is surprising, and encouraging that `Program` ensures we see only these obligations. Nevertheless, it takes a certain familiarity with how it all works in order to untangle these goals, especially the last one.

So, while this approach may work, and indeed predictably so to someone familiar with both the problem and the proof assistant, it remains a problem that the termination and correctness invariants and arguments are now merged into the same Σ-type decorations and proof obligations, respectively. Consequently, it is hard to know for sure which invariant is required for which property, and only by insisting on distinct names and distinct lemmas (used to prove the proof obligation) can one approximate any kind of separation of concerns.

Furthermore, such an approach clearly does not scale: each time one is interested in showing a new property, one has to hack the original definition (to push more and more invariants into the Σ-types) and hack the proof obligations (whose goal is now a bigger conjunction) to insert the proof of the new property.

3.4 Version 2: using the graph relation

The problems suffered by the above approaches arise from the following facts:

- that in this type theory, to reason about a function, that function must have been *already* proved terminating;
- that the only "free" (primitive) induction principle to which one may appeal, corresponds to the structurally decreasing argument, which for a function defined with `Program` and `measure` is a mere implementation detail of the termination proof;
- that even if one could obtain the desired induction principle, it would *still* be polluted with Σ-decoration only needed for termination; this is a necessary consequence of defining the function's type in this way.

But the inductively-defined graph suffers from none of these defects, as observed by other authors before us. By shifting from intensional terms whose termination is internally guaranteed by the type theory, to a formal object that represents a type of "evidence that the function is well-defined", and whose inhabitants must either be constructed (by the user or tactics, when proving well-definedness), or else when available as hypotheses are a witness to well-definedness, we get to "eat our cake and have it too".

For our particular example, we obtain an "unpolluted" version of induction for `reachables_worker` above, namely the canonical principle associated with:

```
Inductive Reachable_rel: forall (visited waiting result: SubsetV), Prop :=
| reachable_empty vs: Reachable_rel vs nil vs
| reachable_cons vs w ws rs:
     Reachable_rel (w :: vs) (rstep vs w ws) rs ->
     Reachable_rel vs (w :: ws) rs.
```

We can now prove soundness and completeness for (input, output) related by `Reachable_rel`:

```
  Lemma sound2 vs ws rs: Reachable_rel vs ws rs ->
    Sound vs -> Sound ws -> Sound rs.

  Lemma complete2 vs ws rs: Reachable_rel vs ws rs ->
    Invariant vs ws -> GComplete vs ws rs.
```

Next, we show that there actually is a function `reachables_worker2` that computes outputs which are related to the inputs by the graph relation. For this, we use `Program Fixpoint` for the final time on this problem:

```
Program Fixpoint reachables_worker2 (visited: SubsetV)
  (waiting: { ws | Termination visited ws })
   {measure measureV visited}: { rs | Reachable_rel visited waiting rs } :=
  match waiting with
  | nil => visited
  | w :: ws => reachables_worker2 (addV w visited) (rstep visited w ws)
  end.
```

This differs from the definition in 3.1 only in that the result type now expresses that the result is related to the inputs by the graph relation. The proof obligations generated are:

- measure decrease, proved as in 3.1;
- preservation of the `Termination` invariant, as in 3.1;
- a last obligation, which could be automated, which essentially observes that the values of the function `reachables_worker2` satisfy the properties expressed by *constructors* of the `Reachable_rel` relation, that is "it has the recursive call structure as specified".

It is now routine to define our final version `reachables2` of the wrapper by

```
Program Definition reachables2: { rs | Specification rs }
  := @reachables_worker2 emptyV start.
```

whose termination obligations are precisely those of soundness and completeness. Now the partial correctness lemmas we proved via the graph come to the fore:

```
  Next Obligation.
  Proof with auto. unfold Specification.
    destruct (@reachables_worker2 emptyV
     (exist (fun l => Termination emptyV l) start reachables2_obligation_1))...
    simpl in *.
    split.
      apply (sound2 r)...
    apply gcomplete_complete...
    apply (complete2 r)...
  Qed.
```

4 Conclusion

We began with a traditional perspective on program verification and the shortcomings of conventional type-theoretic approaches to representing non-structurally recursive programs. We then showed how to progressively refine an approach to defining such functions in CIC using the `Program` machinery, concluding with a development which separates partial correctness out as a purely logical affair, and restricts the use of `Program` to proving termination. This perhaps surprising conclusion comes from paying attention to the separation of concerns, and leads, we believe, to a more abstract, flexible and disciplined approach.

But this is not the last word, since the graph can be derived from the function definition. We hope this paper will inspire the implementors of the `Program` feature in Coq to define this inductive relation automatically from the `Program` construction. We would then not need to do the double work of first defining the function and then its graph with the risk of mistakes such duplication implies.

4.1 Future work

We consider the following extensions to this research, in the definitions and proofs of the algorithms, and in the specific Coq engineering of our strategy:

– we have only considered here a naïve reachability algorithm; nevertheless, by identifying the soundness and completeness lemmas, we are able to drop in any other replacement step function `rstep` having these properties; indeed, one may make the definition of the graph relation *compositional* in the abstract graph of such a step function; we intend to explore this in future publications;
– the algorithm itself is presented here purely concretely in terms of lists, but the correctness proof should be presented as factored through a finite set representation;
– the definition of `reachable` we made is *tail-recursive*, following the iterative prototype of its classical, imperative forebear; so *a priori* it must be run to completion in order to spit out the first element of its result. But a modest tweak to the definition can ensure that at each iteration, the new visited node becomes visible in the output immediately, making the definition a *productive* one. This opens the way to considering the problem of verifying a *co-inductive* definition of `reachable` for infinite graphs;
– going beyond the hypothetical integration of `Program` with the graph machinery, as already envisaged by the first author [2], one could imagine not only systematically synthesising and applying the *canonical* inductive choice of graph induced by a function definition, but further still to consider parametrising the machinery on a *user-supplied* such choice, with the canonical choice as a default.

But such considerations must wait: for now, it is time to terminate the paper!

Acknowledgments The work is partially supported by NWO/BRICKS project "ARPA: Advancing the Real use of Proof Assistants". The first author gratefully acknowledges support of the NWO-funded cluster "DIAMANT".

References

1. McCarthy, J.: A basis for a mathematical theory of computation. In Braffort, P., Hirschberg, D., eds.: Computer Programming and Formal Systems, North-Holland (1963)
2. McKinna, J.: McCarthy-Painter Induction in Epigram. Talk given at the Scottish Theorem Proving (STP) Seminar (July 2003) http://www.cs.ru.nl/~james/stp.pdf.
3. McBride, C., McKinna, J.: The View from the Left. Journal of Functional Programming **14**(1) (2004)
4. Sozeau, M.: Program-ing Finger Trees in Coq. In: Proc. ICFP'07, ACM (2007)
5. Sozeau, M.: Un environnement pour la programmation avec types dépendants. PhD thesis, LRI/Paris XI Orsay (2008)
6. Slind, K.: Reasoning about Terminating Functional Programs. PhD thesis, TU Munich (1999)
7. Balaa, A., Bertot, Y.: Fix-Point Equations for Well-Founded Recursion in Type Theory. In: Proc. TPHOLs'00. Volume 1869 of LNCS., Springer (2000)
8. Barthe, G., Courtieu, P.: Efficient Reasoning about Executable Specifications in Coq. In V. A. Carreno, C. Muñoz and S. Tahar, ed.: Proc. TPHOLs'02. Volume 2410 of LNCS., Springer (2002)
9. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. In Hagiya, M., Wadler, P., eds.: Proc. FLOPS 2006. Volume 3945 of LNCS., Springer (2006)
10. Bove, A.: Programming in Martin-Löf type theory: Unification - A non-trivial example (November 1999) Licentiate Thesis, Chalmers University of Technology.
11. Bove, A., Capretta, V.: Nested General Recursion and Partiality in Type Theory. In Boulton, R.J., Jackson, P.B., eds.: Proc. TPHOLs'01. Volume 2152 of LNCS., Springer (2001)
12. Bove, A.: General Recursion in Type Theory. PhD thesis, Department of Computing Science, Chalmers University of Technology (2002)
13. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
14. Moore, J.S., Zhang, Q.: Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2. In Hurd, J., Melham, T.F., eds.: Proc. TPHOLs'05. Volume 3603 of LNCS., Springer (2005)
15. Gordon, M.J.C., Hurd, J., Slind, K.: Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving. In Geist, D., Tronci, E., eds.: Proc. CHARME'03. Volume 2860 of LNCS., Springer (2003)
16. Nishihara, T., Minamide, Y.: Depth-first search. JAR Archive of Formal Proofs (June 2008) http://afp.sourceforge.net/entries/Depth-First-Search.shtml.

A Coq source `pearl.v`

This appendix (generated with `coqdoc`) is provided for completeness and the convenience of reviewers. The full development, including the `util.v`, `list_util.v`, and `fix_measure_utils.v` utility files, may be obtained from the authors at `www.cs.ru.nl/~james/2009-TPHOLS`. We used the current (8.2) release of Coq.

```
Require Import List.
Require Import util.
Require Import list_util.
Require fix_measure_utils.
Require Import Program.
Require Import Wf_nat.

Set Implicit Arguments.

Module REACHABILITY.
Section definitions.

  Variables (State: Type) (trans: State → State → Prop).

  Inductive reachable: State → State → Prop :=
    | reachable_refl s: reachable s s
    | reachable_next a b c: reachable a b →
        trans b c → reachable a c.

  Lemma reachable_trans a b: reachable a b →
    ∀ c, reachable b c → reachable a c.
  Proof with auto.
    induction 2...
    apply reachable_next with b...
  Qed.

End definitions.
End REACHABILITY.

Hint Constructors reachability.reachable.

Record DiGraph: Type := Build
  { Vertex: Set
  ; Vertex_eq_dec: ∀ (v v': Vertex), decision (v = v')
  ; vertices: list Vertex
  ; vertices_exhaustive: ∀ v, In v vertices
  ; edges: Vertex → list Vertex
  ; edges_NoDup: ∀ v, NoDup (edges v)
  }.

Hint Resolve edges_NoDup.
Hint Immediate edges_NoDup.
Hint Immediate vertices_exhaustive.

Implicit Arguments edges [d].

Section contents.

  Variable G: DiGraph.

  Let Edge (v w: Vertex G): Prop := In w (edges v).

  Let ved := Vertex_eq_dec G. Let subtrV := subtr ved.
Let SubsetV := list (Vertex G).
Let emptyV : SubsetV := []. Hint Unfold emptyV.
Let addV v vs : SubsetV := v :: vs. Hint Unfold addV.

Variable start: SubsetV.
Hypothesis NoDup_start: NoDup start.

Let reachable v: Prop := ∃ s,
  In s start ∧ reachability.reachable Edge s v.

Lemma reachable_start v: In v start → reachable v.
Proof. firstorder. Qed.

Hint Resolve reachable_start.

Lemma reachable_next v: reachable v → ∀ w, Edge v w → reachable w.
Proof with auto.
  intros. repeat destruct H. ∃ x. split...
  apply reachability.reachable_next with v...
Qed.

Definition Sound (ss: SubsetV): Prop := ∀ v, In v ss → reachable v.

Hint Unfold Sound.

Lemma Sound_empty: Sound emptyV. repeat intro. elim H. Qed.
Hint Immediate Sound_empty.

Definition Complete (ss: SubsetV): Prop := ∀ v, reachable v → In v ss.

Definition GComplete (vs ws rs: SubsetV): Prop :=
  closed_under Edge rs ∧ incl ws rs ∧ incl vs rs.

Lemma gcomplete_complete vs: GComplete [] start vs → Complete vs.
Proof. unfold Complete, GComplete.
  intros vs [c [i j]] v [s [b d]].
  induction d; eauto.
Qed.

Definition Specification rs : Prop := Sound rs ∧ Complete rs.

Definition rstep vs w ws :=
  (subtrV (edges w) ((addV w vs) ++ ws)) ++ ws.

Lemma rstep_Sound_lemma vs w ws:
  incl (rstep vs w ws) ((edges w) ++ ws).
Proof with auto. unfold rstep, subtrV.
  repeat intro.
  destruct (in_app_or _ _ _ H)...
  destruct (In_subtr _ _ _ _ H0)...
Qed.

Lemma rstep_GComplete_lemma vs w ws: incl ws (rstep vs w ws).
```

```
Proof with auto.
  repeat intro.
  apply in_or_app...
Qed.

Lemma rstep_Invariant_lemma vs w ws:
  incl (subtrV (edges w) (addV w vs)) (rstep vs w ws).
Proof with auto.
  repeat intro.
  simpl in H.
  unfold rstep. simpl.
  destruct (snd (In_remove _ _ _ _) H).
  destruct (In_subtr _ _ _ _ H0).
  destruct (In_dec ved a ws)...
  apply in_or_app.
  left.
  apply In_remove'...
  apply subtr_In...
  intro.
  destruct (in_app_or _ _ _ H4)...
Qed.

Let neighbours := flat_map (@edges G).

Definition Invariant vs ws: Prop := incl (neighbours vs) (ws ++ vs).

Lemma Invariant_empty l: Invariant emptyV l.
Proof. unfold Invariant. intuition. Qed.

Hint Immediate Invariant_empty.

Lemma Invariant_preserved vs w ws:
 Invariant vs (addV w ws) →
 Invariant (addV w vs) (rstep vs w ws).
Proof with auto. unfold Invariant.
  intros.
  unfold neighbours in ×. simpl.
  apply incl_app.
    repeat intro.
    destruct (In_dec ved a (addV w vs))...
    apply in_or_app.
    left.
    apply rstep_Invariant_lemma...
    apply subtr_In...

  apply incl_tran with ((addV w ws) ++ vs)...
  apply incl_app.
    apply incl_cons...
    eapply incl_appr...
    unfold addV...
    apply incl_appl.
    apply incl_appr...

    apply incl_appr...
    unfold addV...
    unfold incl. eauto.
  Qed.

  Lemma Invariant_closed rs: Invariant rs emptyV → closed_under Edge rs.
  Proof with auto.
    intros. apply closed_by_flat_map_incl...
  Qed.

  Definition measureV (vs: SubsetV): nat := length (subtrV (vertices G) vs).

  Lemma measureV_decrease ws w vs:
    Disjoint (w :: ws) vs → measureV (w :: vs) < measureV vs.
  Proof.
    intros. unfold measureV. apply remove_length_lt.
    eapply subtr_In. eauto. apply (fst (Disjoint_cons H)).
  Qed.

  Definition Termination (vs ws: SubsetV): Prop :=
    NoDup ws ∧ Disjoint ws vs.

  Lemma Termination_start: Termination emptyV start.
  Proof. split; auto. intro. intuition. Qed.
  Hint Resolve Termination_start.

  Lemma NoDup_rstep vs w ws:
    NoDup (addV w ws) → NoDup (rstep vs w ws).
  Proof with auto.
    intros.
    inversion_clear H.
    apply NoDup_app...
    apply NoDup_subtr...
    repeat intro.
    destruct (snd (In_remove ved _ _ _) H).
    destruct (In_subtr ved _ _ _ H3)...
    destruct (not_In_app _ _ _ H6)...
  Qed.

  Lemma Disjoint_rstep vs w ws:
    NoDup (addV w ws) → Disjoint (addV w ws) vs → Disjoint (rstep vs w
ws) (addV w vs).
  Proof with auto.
    intros.
    inversion_clear H.
    destruct (Disjoint_cons H0).
    unfold rstep, subtrV.
    repeat intro.
  destruct (in_app_or _ _ _ H4)...
  destruct (In_subtr _ _ _ _ H6)...
  destruct H5.
   subst...
   destruct (H3 x)...
Qed.

Lemma Termination_preserved vs w ws:
  Termination vs (addV w ws) →
  Termination (addV w vs) (rstep vs w ws).
Proof with auto. unfold Termination.
  intros.
  destruct H.
  split.
    apply NoDup_rstep...
  apply Disjoint_rstep...
Qed.

Program Fixpoint reachables_worker (visited: SubsetV)
  (waiting: { ws | Termination visited ws })
  {measure measureV visited}: SubsetV :=
  match waiting with
  | nil ⇒ visited
  | w :: ws ⇒ reachables_worker (addV w visited) (rstep visited w ws)
  end.

Next Obligation. Proof with auto. destruct H. apply measureV_decrease
with ws... Qed.
Next Obligation. Proof with auto. apply Termination_preserved... Qed.

Program Definition reachables: SubsetV
  := @reachables_worker emptyV start.

Lemma rw_isEta: isEta _ reachables_worker. apply isEta_wit. Defined.
Definition reachables_worker0 := unEta rw_isEta.

Implicit Arguments reachables_worker0 [].

Lemma isFix_measure_sub:
  fix_measure_utils.isFix_measure_sub measureV _ reachables_worker0.
Proof.
  unfold reachables_worker0. simpl.
  apply fix_measure_utils.show_isFix_measure_sub.
Defined.

Lemma reachables_worker0_ind_aux
 (P: ∀ (vs: SubsetV) (ws: {l | Termination vs l}), SubsetV → Prop)
 (Pbase: ∀ vs a, P vs (exist _ emptyV a) vs)
 (Prec: ∀ vs w ws p rs,
    P (w :: vs) (exist _ (rstep vs w ws) p) rs → ∀ mp,
    P vs (exist _ (w::ws) mp) rs):
  ∀ visited waiting, P visited waiting (@reachables_worker0 visited waiting).
Proof with auto.
  do 4 intro.
  pattern visited, (reachables_worker0 visited).
  apply (fix_measure_utils.rect isFix_measure_sub).
  clear visited.
  intros.
  rename x into visited.
  destruct waiting.
  rename x into waiting.
  destruct waiting; simpl...
  apply (Prec _ _ _ _ _ (X _ _ (exist _ (rstep visited v waiting) _))).
Qed.

Lemma reachables_worker0_ind (P: ∀ (vs ws rs: SubsetV), Prop)
  (Pbase: ∀ vs, P vs emptyV vs)
  (Prec: ∀ vs w ws rs, P (addV w vs) (rstep vs w ws) rs →
    P vs (w :: ws) rs):
  ∀ vs ws, P vs (‘ws) (reachables_worker0 vs ws).
Proof with auto.
  do 5 intro. pattern vs, ws, (reachables_worker0 vs ws).
  apply reachables_worker0_ind_aux...
  simpl.
  intros.
  clear mp. apply Prec...
Qed.

Lemma sound vs ws: Sound vs →
  Sound (‘ws) → Sound (reachables_worker0 vs ws).
Proof with simpl; auto.
  do 2 intro.
  pattern vs, (‘ws), (reachables_worker0 vs ws).
  apply reachables_worker0_ind; unfold Sound...
  intros.
  apply H... intuition.
  intros.
  destruct (in_app_or _ _ _ (rstep_Sound_lemma _ _ _ _ H3))...
  apply reachable_next with w...
Qed.

Lemma complete vs ws: Invariant vs (‘ws) →
  GComplete vs (‘ws) (reachables_worker0 vs ws).
Proof with unfold emptyV, addV; simpl; auto.
  do 2 intro.
  pattern vs, (‘ws), (reachables_worker0 vs ws).
  apply reachables_worker0_ind; unfold GComplete; intros.
    intuition.
    apply Invariant_closed...
 unfold emptyV...
  destruct H. apply Invariant_preserved...
  unfold addV in H1; destruct H1.
  intuition...
    apply incl_cons...
    apply incl_tran with (rstep vs0 w ws0)...
    apply rstep_GComplete_lemma.
  repeat intro...
Qed.

Program Definition reachables0: { rs | Specification rs }
  := @reachables_worker0 emptyV start.

Next Obligation.
Proof with auto. unfold Specification.
  split.
    apply sound...
  apply gcomplete_complete...
  apply complete...
Qed.

Program Fixpoint reachables_worker1
  (visited: { vs | Sound vs })
  (waiting: { ws | Sound ws ∧ Termination visited ws ∧ Invariant visited ws})
    {measure measureV visited}:
      { rs | Sound rs ∧ GComplete visited waiting rs } :=
  match waiting with
  | nil ⇒ visited
  | w :: ws ⇒ reachables_worker1 (addV w visited) (rstep visited w ws)
  end.

Next Obligation.

Proof with auto.
  destruct H; destruct H1...
  repeat split...
  apply Invariant_closed...
Qed.

Next Obligation.

Proof. unfold Sound in ×. simpl. intuition. subst. auto. Qed.

Next Obligation.
Proof. apply measureV_decrease with ws. firstorder. Qed.

Next Obligation.

Proof with simpl; auto.
```

```
  destruct H. destruct H1.
  destruct (Termination_preserved H1).
  destruct H1.
  inversion_clear H1.
  unfold Sound.
  repeat split; intros...
    destruct (in_app_or _ _ _ (rstep_Sound_lemma visited w ws _ H1))...
    apply reachable_next with w...
  apply Invariant_preserved...
Qed.

Next Obligation. Proof with auto.
  match goal with
  [ ⊢ context[reachables_worker1 ?a ?b] ] ⇒
    destruct (reachables_worker1 a b)
  end.
  simpl in ×. clear reachables_worker1. subst.
 unfold addV, GComplete, Invariant in ×.
    intuition...
    apply incl_cons...
    apply incl_tran with (rstep visited w ws)...
    apply rstep_GComplete_lemma.
  repeat intro...
Qed.

Program Definition reachables1: { rs | Specification rs }
  := @reachables_worker1 emptyV start.

Obligation Tactic := idtac.

Next Obligation.
Proof with intuition; auto. unfold Specification.
  match goal with
  [ ⊢ context[reachables_worker1 ?a ?b] ] ⇒
    destruct (reachables_worker1 a b)
  end.
  simpl in ×.
  split...
  apply gcomplete_complete...
Qed.

Obligation Tactic := program_simpl.

Inductive Reachable_rel: ∀ (visited waiting result: SubsetV), Prop :=
  | reachable_empty vs: Reachable_rel vs nil vs
  | reachable_cons vs w ws rs:
      Reachable_rel (w :: vs) (rstep vs w ws) rs →
      Reachable_rel vs (w :: ws) rs.

Hint Constructors Reachable_rel.
```

```
Lemma sound2 vs ws rs: Reachable_rel vs ws rs →
  Sound vs → Sound ws → Sound rs.
Proof with simpl in ×; auto.
  unfold Sound.
  induction 1...
  intros. apply IHReachable_rel...
    intros. destruct H3...
  intros. destruct (in_app_or _ _ _ (rstep_Sound_lemma _ _ _ _ H3))...
  apply reachable_next with w...
Qed.

Lemma complete2 vs ws rs: Reachable_rel vs ws rs →
  Invariant vs ws → GComplete vs ws rs.
Proof with simpl in ×; auto. unfold GComplete.
  induction 1; intros.
    split...
    apply Invariant_closed...
  destruct IHReachable_rel. apply Invariant_preserved...
  intuition; repeat intro...
  destruct H2. subst...
  apply H3.
  apply (rstep_GComplete_lemma vs w ws)...
Qed.

Program Fixpoint reachables_worker2
  (visited: SubsetV)
  (waiting: { ws | Termination visited ws })
   {measure measureV visited}: { rs | Reachable_rel visited waiting rs } :=
  match waiting with
  | nil ⇒ visited
  | w :: ws ⇒ reachables_worker2 (addV w visited) (rstep visited w ws)
  end.

Next Obligation. Proof with auto. destruct H. apply measureV_decrease
with ws... Qed.
Next Obligation. Proof with auto. apply Termination_preserved... Qed.
Next Obligation. Proof with auto. apply sig_self. subst... Qed.

Program Definition reachables2: { rs | Specification rs }
  := @reachables_worker2 emptyV start.

Next Obligation.
Proof with auto. unfold Specification.
  destruct (@reachables_worker2 emptyV
     (exist (fun l ⇒ Termination emptyV l) start reachables2_obligation_1))...
  simpl in ×.
  split.
    apply (sound2 r)...
  apply gcomplete_complete...
  apply (complete2 r)...
Qed.

Variable Rstep : ∀ (visited : SubsetV)(w: Vertex G)(waiting step: SubsetV),
Prop.
Hypothesis Rstep_Sound : ∀ vs w ws S, ∀ r: Rstep vs w ws S, incl S ((edges
w) ++ ws).
Hypothesis Rstep_GComplete : ∀ vs w ws S, ∀ r: Rstep vs w ws S, incl ws
S.
Hypothesis Rstep_Invariant : ∀ vs w ws S, ∀ (r: Rstep vs w ws S),
  incl (neighbours (addV w vs)) (S ++ (addV w vs)).
Hypothesis Rstep_Termination : ∀ vs w ws S, ∀ (r: Rstep vs w ws S),
    Termination vs (addV w ws) → Termination (addV w vs) S.

Inductive Reachable_abs
  (Rstep : ∀ (visited: SubsetV)(w: Vertex G)(waiting step: SubsetV), Prop)
: ∀ (visited waiting result: SubsetV), Prop :=
  | reachable_abs_empty vs: Reachable_abs Rstep vs nil vs
  | reachable_abs_cons vs w ws ss rs : Rstep vs w ws ss →
      Reachable_abs Rstep (w :: vs) ss rs →
      Reachable_abs Rstep vs (w :: ws) rs.

Hint Constructors Reachable_abs.

Lemma sound_abs vs ws rs: Reachable_abs Rstep vs ws rs →
  Sound vs → Sound ws → Sound rs.

Proof with simpl in ×; auto.
  unfold Sound.
  induction 1...
  intros. apply IHReachable_abs...
    intros. destruct H4...
  intros. destruct (in_app_or _ _ _ (@Rstep_Sound _ _ _ _ H _ H4))...
  apply reachable_next with w...
Qed.

Lemma complete_abs vs ws rs: Reachable_abs Rstep vs ws rs →
  Invariant vs ws → GComplete vs ws rs.
Proof with simpl in ×; auto. unfold Invariant, GComplete.
  induction 1; intros.
    split...
    apply Invariant_closed...
  destruct IHReachable_abs. apply (@Rstep_Invariant _ _ _ _ H)...
  destruct H3.
  intuition; repeat intro...
   destruct H5. subst...
   eapply H3... apply (Rstep_GComplete H)...
Qed.

End contents.
```