Stop when you are Almost-Full

Adventures in constructive termination

Dimitrios Vytiniotis
Microsoft Research, Cambridge, U.K.
dimitris@microsoft.com

Thierry Coquand
University of Gothenburg
coquand@chalmers.se

Abstract

Disjunctive well-foundedness (used in Terminator), size-change termination, and well-quasi-orders (used in supercompilation and term-rewrite systems) are examples of techniques that have been successfully applied to automatic proofs of program termination and online termination testing, respectively. Although these works originate in different communities, there is an intimate connection between them – they rely on closely related principles and both employ similar arguments from Ramsey theory. At the same time there is a notable absence of these techniques in programming systems based on constructive type theory. In this paper we'd like to highlight the aforementioned connection and make the core ideas widely accessible to theoreticians and Coq programmers, by offering a Coq development which culminates in some novel tools for performing induction. The benefit is nice composability properties of termination arguments at the cost of intuitive and lightweight user obligations. Inevitably, we have to present some Ramsey-like arguments: Though similar proofs are typically classical, we offer an entirely constructive development standing on the shoulders of Veldman and Bezem, and Richman and Stolzenberg.

1. Introduction

Program termination has always been an exciting subject among researchers, dating back to the early days of computing. The reason is because program termination is at the same time *important* for software reliability, and *difficult* for general classes of programs. Despite the difficulties, however, several research communities have managed to make good progress in termination-related problems.

Over the recent years, the so-called *transition invariants* [20] method has been an extremely successful approach for automatic proofs of program termination, leading to industrial-strength tools, such as Terminator [7]. *Size-change termination* [15, 12, 24] is another very successful methodology for automatic proofs of program termination. In the core of both works lies a formal argument from Ramsey theory [11].

Furthermore, research on online termination testing [16] and supercompilation [25] has for a while been using termination testing criteria for function reductions and inlining based on *well-quasi-orders*, often employing Ramsey-like arguments to form more complex termination testing criteria from simpler ones.

There is an intimate connection between these worlds, and a notable absence of similar techniques to help programmers *prove* the termination or totality of their fixpoint definitions in Coq and – more generally – in systems based on constructive type theory. In this paper we'd like to highlight this connection and make the core ideas widely accessible to theoreticians and Coq programmers, by offering a Coq development which introduces some novel variations of induction principles. Inevitably, we have to present some Ramsey-like arguments: Though similar proofs are typically classical, we offer a constructive development in the footsteps of Veldman and Bezem [28, 9], and Richman and Stolzenberg [22].

Specifically, our contributions with this paper are:

- We introduce a novel mechanism for type-based termination, that of *almost-full* relations (Section 2), which is a weaker version of the more traditional well-quasi-orders, originating in intuitionistic mathematics.
- We formally explain the connection between almost-full relations and well-founded relations (Section 3), and prove a new induction theorem based on almost-full relations. (Section 3.1)
- We demonstrate that, unlike well-founded relations, almost-full relations compose nicely to form other almost-full relations (Section 4). In this context, of particular interest is a construction which constitutes a contribution on its own: a short proof of (the constructive version of) Ramsey's theorem (for binary relations). We give examples of composing almost-full relations to show termination. Thanks to the composability of almost-full relations, the user obligations from the new induction principles typically involve intuitive (and amenable to automation) relation inclusion lemmas instead of proofs about accessibility predicates.
- We can use our method to show examples from *size-change* termination that go beyond simple lexicographic orders (Section 5). We show that the size-change principle can be proved from our induction principle.
- Connecting our work with further work on automatic termination proofs, we show how the almost-full induction principle can be instantiated to derive the Terminator rule (which is based on the so-called *disjunctive well-foundedness*). (Section 6)
- We discuss design decisions and other variations of fixpoint rules, including mutual induction and a convenient induction principle that resembles *inlining*, as well as the computational content of our development (Section 7). Finally we present related work (Section 8) and outline further directions for research. (Section 9)

Our accompanying development does not make use of any "non-standard" axioms in Coq (such as classical facts, proof-irrelevance, or even the more benign functional extensionality). It builds under Coq 8.3pl2.

The new induction principles proposed in this paper are not-necessarily more expressive or easier to use than other (particularly recent [6, 13, 27]) related work – this is a topic that deserves further investigation, engineering, and potentially automation support. On the other hand, the induction principles that we propose here are quite amenable to automation and quite pleasant to use due to the composability of almost-full relations and the nature of the user obligations that arise. Apart from contributing to Coq's large arsenal of recursion-encoding techniques [4, 5, 17, 27, 6], the other significant contribution of this article is to bring together ideas from different research communities in a type-theoretic framework.

2. Well-quasi-orders and almost-full relations

The starting point for this exploration will be *online termination testing*. Online termination testing is concerned with the following problem: Assume that we monitor the execution of a program by means of observing the state or the arguments passed down in recursive calls; can we raise an error as soon as we detect that the program might be diverging? The requirement for a sound termination tester is that we *must* raise an error if the program is indeed divergent. Conversely, we can't expect in general to raise an error *only* if the program is divergent but we'd like our online termination tester to be as lenient as possible.

Assume now that the observed state of a program forms a sequence of values $s_1, s_2, \ldots$ – in effect we'd like to detect if that sequence s could be infinite. There is a very natural mathematical definition that can help us here, and that is the notion of a *well-quasi-order* (WQO):

Definition 1 (Well-quasi-order)**.** For some set `X`, a binary relation $\preceq$ of type `X → X → Prop` is a well quasi order if (i) it is transitive and (ii) for every infinite sequence s of elements of `X` it is the case that there exist i and j with $i < j$ such that $s_i \preceq s_j$.

Why is this helpful? Consider the following online termination tester which accepts a user-provided WQO $\preceq$ as input: We will keep a record of all previous values we have observed and every time a new value s_{new} appears, we will check if for some old value s_{old} it is $s_{old} \preceq s_{new}$. If this is true then we will raise an error, otherwise we will record s_{new} in our history and wait for the next value. Now, if the sequence was infinite then we definitely know that we will raise an error at some point (because $\preceq$ is a WQO). Of course, conservatively, we might raise an error even when the sequence is not infinite because the WQO provided was too conservative.

Let us demonstrate this with an example. On the type `nat`, the relation $\leq$ (`le` in Coq) is a WQO: Think of any infinite sequence of natural numbers – at some point we will meet a natural number which is greater or equal than some previous one. Hence, our online termination tester would raise an error for the following sequence:

$$10, 7, 6, 4, 1, 5, 4, 3, 3, 3, \ldots$$

as soon as it encountered the element 5, since $4 \leq 5$. Of course, if the sequence is actually finite (e.g. it ends after a hundred 3 values) then, too bad: our WQO has been too conservative and we should have used a more lenient one.

The merits of WQOs for online termination testing have been discussed in previous work [16] so we will not go into details here. Their main advantage is that they can form extremely lenient termination tests by combining simpler WQOs (at the cost of having to record big portions of history).

2.1 Almost-full relations

The mechanism described above is by now well-established for online termination testing, so it is quite natural to ask how it would look in type theory and check if it can be used to *prove* termination, in addition to testing for termination.

Surprisingly, it turns out that a certain kind of relations that satisfy property (ii) in the definition of WQOs have been proposed by mathematicians in an entirely different domain: the development of an intuitionistic version of Ramsey theory [28]. These are the *almost-full* (AF) relations (term coined by Wim Veldman), and for the rest of this paper we will focus on *binary* AF relations.

An AF relation is a powerful *inductive* characterization of relations that satisfy property (ii) above. To introduce AF relations we follow the short note [9], which uses the auxiliary definition of *well-founded trees* on a set `X`:[1]

```
Inductive WFT (X : Set) : Set :=
  | ZT : WFT X
  | SUP : (X → WFT X) → WFT X.
```

An instructive way to understand a well-founded tree is as a set of winning strategies in a game. At each point either the game has finished and we won (`ZT`) or (in the case of `SUP`) the opponent can provide a next value of type `X` and we are asked to come up with a next move. Because the tree is inductive, all such "moves" end up in a winning position.

But, the winning strategies of which game exactly does a well-founded tree represent? Well, we wrote before that we are interested in relations for which every infinite sequence contains two values that are related. Let us define when a well-founded tree of type `WFT X` "secures" a binary relation `A : X → X → Prop`:

```
Fixpoint SecureBy (X:Set) (A : X→ X→ Prop)
                  (p : WFT X) : Prop :=
 match p with
 | ZT ⇒ ∀ x y, A x y
 | SUP p ⇒
    ∀ x, SecureBy (fun y z ⇒ A y z ∨ A x y) (p x)
 end.
```

Now if we know that `SecureBy R p` and the tree `p` is `ZT` then all elements of the domain `X` are related, and hence the tree is a witness that every infinite sequence has two related elements. If the tree is `SUP p` then, when presented with a new value `x`, either `x` *itself* is related to some future element `y`, or we have to keep going to find the two related elements in the future. This explains the relation `fun y z ⇒ A y z ∨ A x y` used in the recursive call. Because `p` is inductive, we cannot go on for ever. In the leaves we will have established that some values along the way were related by `A`!

This leads to the definition of an AF relation, namely one that is "secured" in the way we have defined by some well-founded tree:

```
Definition almost_full (X:Set) (A : X→ X→ Prop) :=
  ∃ p, SecureBy A p.
```

We can now prove condition (ii) of the definition of WQOs:

```
Lemma sec_binary_infinite_chain :
    ∀ (X:Set) (p : WFT X) R (f : nat → X) (k : nat),
    SecureBy R p →
    ∃ n, ∃ m, (n > m) ∧ (m >= k) ∧ R (f m) (f n).
```

The sketch of the proof is based on the discussion above. We use function `f` as an infinite sequence. Imagine a cursor at point `k` of the infinite sequence. Then we have two cases: Either the current tree is `ZT` in which case we can simply take the elements at positions `k` and `k+1`, or the tree is `SUP p`. In the latter case, by induction, we are guaranteed to find two related elements in the future, or an element which is related to the element in position `k`. In both cases we are done! As a corollary:

```
Corollary af_inf_chain (X : Set) (R : X → X → Prop):
   almost_full R →
   ∀ (f : nat → X), ∃ n, ∃ m, (n > m) ∧ R (f m) (f n).
```

[1] But see Section 7 for a discussion on an alternative possible formalization.

It is interesting to observe that corollary `af_inf_chain` is quite analogous to the "no infinite descending chain" property which can be proved intuitionistically from Coq's inductive definition of well-founded relations (based on accessibility predicates) [4]. The converse of `af_inf_chain` holds classically but Veldman and Bezem [28] show that there exists a recursive counterexample and so does not hold in type theory. This makes it impossible to use property (ii) as the very defining property of AF relations (instead of our inductive characterization) because that alternative definition cannot be used for induction (see Theorem `wf_from_af` in Section 3.1).

Notice also that we've stopped worrying about the transitivity condition – indeed we are not going to need it! Some of the proofs in later parts of the paper would be simpler (and they are, in related work [18]) but essentially all interesting properties of WQOs can be proved on AFs without requiring transitivity.

3. Well-founded vs. almost-full relations

To build up some more intuitions about AF relations, we now turn to the connection between AF relations and well-founded relations. A well-founded relation can be constructively characterized as a relation where every element in its domain is *accessible*. The corresponding (standard) Coq definitions are:

```
Inductive Acc (A:Type) (R:A→ A→ Prop) (x:A) : Prop :=
  Acc_intro : (∀  y : A, R y x → Acc R y) → Acc R x.
Definition well_founded :=
  fun (A:Type) (R:A → A → Prop) ⇒ ∀ a:A, Acc R a.
```

Coq comes with a library for constructing well-founded relations as well as proofs that several relations on commonly used datatypes are well-founded, the $<$ relation on `nat` being the simplest example.

It is easy to construct AF relations from *decidable* well-founded relations: If we are given a decidable WF relation `R` then we will show next that the relation `fun x y ⇒ not (R y x)` is AF. This will enable us to re-use Coq libraries and lemmas for WF relations in developments for AF relations. Let us define the operation `af_tree_iter`, which builds a well-founded tree by iteration on an accessibility predicate:

```
Definition dec_rel (X:Set) (R:X→ X→ Prop) :=
  ∀ x y, {not (R y x)} + {R y x}.

Fixpoint af_tree_iter (X:Set) (R : X → X → Prop)
         (decR : dec_rel R) (x:X) (accX : Acc R x) :=
 match accX with
 | Acc_intro f ⇒ SUP (fun y ⇒
     match decR x y with
       | left _ ⇒ ZT X
       | right Ry ⇒ af_tree_iter decR (f y Ry)
     end)
 end.
```

The purpose of this construction is to create a tree which secures `fun x y ⇒ not (R y x)`. We accept as input an `x` which is accessible and we iterate on the accessibility predicate. We create a `SUP` node. When we are presented with a "next" element `y` we check whether `R y x` or not using the decidability predicate `decR`. If not, then we have immediately found both the elements we needed, the relation is secured, and we can return `ZT`. If on the other hand it is `R y x` then we may simply continue iterating!

With this definition, it is not difficult to derive the following.

```
Definition af_tree (X:Set) (R : X → X → Prop)
 (wfR : well_founded R) (decR : dec_rel R) : X → WFT X.
```

```
Lemma secure_from_wf :
 ∀ (X:Set) (R : X → X → Prop)
 (wfR : well_founded R) (decR : dec_rel R),
 SecureBy (fun x y ⇒ not (R y x))
         (SUP (af_tree wfR decR)).

Corollary af_from_wf (X:Set) (R : X → X → Prop) :
  well_founded R →
  dec_rel R → almost_full (fun x y ⇒ not (R y x)).
```

Corollary `af_from_wf` allows us to go from a decidable well-founded relation to an AF. With this principle, and taking into account that $<$ is decidable and a total order, we can actually prove:

```
Corollary leq_af : almost_full le.
```

since $\forall xy, x \leq y \leftrightarrow \neg(y < x)$ on natural numbers.

3.1 From almost-full to well-founded relations

So far AF relations appear to be a funny flipped-over version of Coq's WF relations, so it's time we saw how can they be used to prove termination. The key intuition comes from online termination testing with WQOs. Recall that a WQO-based termination checker takes a WQO $\preceq$ and a "history" of past values and when presented with a new value checks whether some old value from the history is related to this new value.

Think now of the relation `T : X → X → Prop` which relates all adjacent values s_{i+1} and s_i (and only those) in the input sequence. This is often called the *transition relation* of the program that generates this sequence. As a convention we will be using the first argument of `T` as the "next" value and the second as the "current" value (so that we have $T\ s_{i+1}\ s_i$ for every i). The termination test that our WQO-based checker effectively implements is that:

$$T^+ \cap (\preceq)^{-1} = \emptyset$$

where T^+ is the transitive closure of T and $(\preceq)^{-1}$ is just the inverse of $\preceq$. No infinite sequence can pass this test, because an infinite sequence will necessarily have elements related by $\preceq$! Put it another way, if the test succeeds the transition relation cannot have infinite chains – well, it is well-founded!

Generalizing our intuition from transition relations to arbitrary relations, and weakening the assumptions from WQOs to AF relations, the following lemma is the most important result of this paper, hence we put it in a big box:

```
Lemma wf_from_af :
  ∀ (X:Set) (p : WFT X)
  (R : X → X → Prop) (T : X → X → Prop),
  (∀  x y, clos_trans_1n X T x y ∧ R y x → False)
   → SecureBy R p → well_founded T.
```

In the `wf_from_af` lemma, `clos_trans_1n X T` is just the transitive closure of `T`, as defined in Coq's standard library.

Using `wf_from_af` we can derive a simple lemma for transitive AFs, which are WQOs:

```
Lemma wf_from_wqo :
  ∀ (X:Set) (p : WFT X) (R : X → X → Prop),
        transitive X R → SecureBy R p →
        well_founded (fun x y ⇒ R x y ∧ not (R y x)).
```

For instance, for the $\leq$ relation on natural numbers, it is clearly the case that $\lambda xy.x \leq y \wedge \neg(y \leq x)$ is WF. This relation is simply $<$.

Of course the proof of `wf_from_af` is done in a constructive setting, no infinite descending chains are involved, and hence the mathematically curious reader may wonder how the proof of this theorem goes. We present it next – the non-curious reader can skip directly to Section 3.2.

The proof of lemma `wf_from_af` is done through the following generalization:

```
Lemma acc_from_af:
  ∀ (X:Set) (p : WFT X) (R : X → X → Prop)
  (T : X → X → Prop) y,
  (∀ x z, clos_refl_trans X T z y →
              clos_trans_1n X T x z ∧ R z x → False)
  → SecureBy R p → Acc T y.
```

To understand the generalization in lemma `acc_from_af` it is easier to think again the special case of `T` as a "transition relation" of a program. With lemma `acc_from_af` we focus on a particular element `y` and show that it is accessible. Instead of assuming that the whole transitive closure of T does not intersect with (the inverse of) `R`, we will assume the same property only for the part of the transitive closure that "follows" `y` in the sequence. We proceed by induction on `p`:

- If `p = ZT` then all elements are related by `R`. It suffices to show that all elements `z` for which `T z y` are accessible (by the `Acc_intro` constructor). But then `clos_refl_trans X T y y` and also `clos_trans_1n X T z y` and `R y z` which is a contradiction, so this case is done.
- If `p = SUP w` then, again, it suffices to show that all elements `z` for which `T z y` are accessible (by the `Acc_intro` constructor). We know that `w y` secures the relation
  `Ry := fun y0 z0 ⇒ R y0 z0 ∨ R y y0`
  Thus, we may apply the induction hypothesis for `z`, instantiating `R` to `Ry`. To finish the case we need to show that for any `x` and `w` such that `clos_refl_trans X T w z` it is impossible to have `clos_trans_1n X T x w ∧ (R w x ∨ R y w)`, given that it is impossible to have `clos_trans_1n X T x w ∧ R w x` for all elements `x` and `w` with `clos_refl_trans X T w y`. Let us pick any `w` "following" `z`. Since `clos_refl_trans X T w z` and `T z y` it must be that `clos_refl_trans X T w y`. So it cannot be that `clos_trans_1n X T x w ∧ R w x`. The only case we have to rule out is when `clos_trans_1n X T x w` and `R y w`: But here we have `clos_refl_Trans X T y y` and `clos_trans_1n X T w y` and `R y w`, which is again a contradiction.

Deriving Lemma `wf_from_af` from this generalization is a trivial task, since every element is related to itself in the reflexive transitive closure of `T`, hence every element is accessible, thus `T` is well-founded.

3.2 A new induction principle

If we can use lemma `wf_from_af` to form WF relations, then we can surely use it to perform induction. The following theorem `af_induction` demonstrates a new induction principle, based on `wf_from_af`.

```
Theorem af_induction:
   ∀ (A:Set) (T : A → A → Prop) (R : A → A → Prop),
   almost_full R →
   (∀ x y, clos_trans_1n A T x y ∧ R y x → False) →
   ∀ P : A → Set,
         (∀ x, (∀ y, T y x → P y) → P x)
          → ∀ a, P a.
```

Intuitively `T` is the relation between the argument in the "next" recursive call (`y`), and the previous (`x`) and we are simply requiring that the transitive closure of `T` has an empty intersection with (the inverse of) some AF relation `R`.

Hence, when using AF induction, the programmer must (i) provide an AF relation, (ii) show the emptyness of the intersection, and (iii) provide a functional for the recursive call.

Example 1. Let's see this induction principle in action with a very simple example that computes Fibonacci numbers:[2]

```
Definition fib : nat → nat.
apply af_induction with (T := lt) (R := le).
(* (i) Prove <= is AF *)
apply leq_af.
(* (ii) Prove intersection emptyness *)
intros x y (CT,H). induction CT; repeat firstorder.
(* (iii) Give the functional *)
refine (fun x ⇒
  match x as w return (∀ y, y < w → nat) → nat with
    | 0 ⇒ fun frec ⇒ 1
    | 1 ⇒ fun frec ⇒ 1
    | (S (S x)) ⇒ fun frec ⇒ frec (S x) _ + frec x _
  end); firstorder.
Defined.
```

As a remark, to ensure that computation does not get stuck we have used `Defined` instead of `Qed` (and we do so in most of our development), which makes the various lemmas and definitions transparent for computation purposes.

It is worth contrasting `af_induction` with Coq's standard well-founded induction, and Figure 1 presents our new induction principle side-to-side to Coq's standard WF induction. Notably, `af_induction` takes both `T` and an AF relation `R`, the empty intersection requirement, and the functional for the recursion, whereas `well_founded_induction` only requires a proof that `T` is well-founded.

Summary So far we have generalized and proved the underlying principle behind WQO-based online termination testing and revealed the connections between AF relations and WF relations. We have used this underlying principle to derive a simple AF-based induction principle. From a programmer's perspective this new principle does not *yet* seem to have made things nicer really, as the user now has to prove two preconditions instead of one. So it's time we move on to the benefits of using AF relations that we promised to deliver in the introduction.

4. Constructions on AF relations

The nicest characteristic of AF relations is their *remarkable composability*. In this section we will show various results that substantiate this claim: the union of AF relations is AF (Section 4.1) and the intersection of AF relations is AF (Section 4.2). We also show type-based compositions (Section 4.3): how to use map-like operations, how to create AF relations for products, tagged unions (sums), and finite types (such as `bool`). Together with our results from Section 3, which can be used to give us "ground" AF relations from existing WF relations, this section presents a powerful toolkit for the `af_induction` user.

4.1 AF unions

Unions are not particularly difficult. If we are presented with an infinite sequence in which there exist two related elements by relation `A` then obviously these two elements are also related by the relation `A ∪ B`.

[2] Of course `af_induction` is a pretty contrived way to write this simple function but we'd like to demonstrate the three steps with the simplest possible example before we dive in more complex examples.

AF induction	WF induction

<pre>af_induction
  : ∀ (A : Set) (T R : A → A → Prop),
    almost_full R →
    (∀ x y : A, clos_trans_1n A T x y ∧ R y x → False) →
    ∀ P : A → Set,
    (∀ x : A, (∀ y : A, T y x → P y) → P x) →
    ∀ a : A, P a</pre>
	<pre>well_founded_induction
	  : ∀ (A : Set) (T : A → A → Prop),
	    well_founded T →
	    ∀ P : A → Set,
	    (∀ x : A, (∀ y : A, T y x → P y) → P x) →
	    ∀ a : A, P a</pre>

Figure 1: AF vs WF induction principles

More generally, it takes a straightforward induction on well-founded trees to prove the following lemma:

```
Lemma sec_strengthen:
 ∀ (X:Set) (p : WFT X) (A B : X → X → Prop),
 (∀ x y, A x y → B x y) → SecureBy A p → SecureBy B p.
```

From which the following two results follow:

```
Lemma sec_union (X:Set) (A:X→ X→ Prop) (B:X→ X→ Prop):
 ∀ p, SecureBy A p →
   SecureBy (fun x y ⇒ A x y ∨ B x y) p.

Corollary af_union:
 ∀ (X:Set) (A:X→ X→ Prop) (B:X→ X→ Prop),
 almost_full A →
 almost_full (fun x y ⇒ A x y ∨ B x y).
```

4.2 AF intersections

Intersections are much harder. Imagine that we have AF relations `A` and `B`. If we are presented with an infinite sequence then we definitely know that `A` relates some elements in the sequence, and `B` relates some elements in the sequence, but are there any elements that are *simultaneously* related by `A` and `B`? Remarkably, the answer is affirmative. A generalization of this theorem to k-ary AF relations is often called the “intuitionistic version of Ramsey’s theorem” [28].[3]

Here we focus on the binary case, following and simplifying the setup in [28, 9]. The idea of the proof is that, given two well-founded trees that secure relations `A` and `B` respectively, we will construct another one that secures their intersection. This construction involves three stages.

- First, we define the `oplus_nullary` function below:

  ```
  Fixpoint oplus_nullary (X:Set) (p:WFT X) (q:WFT X) :=
    match p with
    | ZT ⇒ q
    | SUP f ⇒ SUP (fun x ⇒ oplus_nullary (f x) q)
    end.
  ```

  The function `oplus_nullary` secures intersections of nullary predicates, which is shown with the following lemma:

  ```
  Lemma oplus_nullary_sec_intersection:
    ∀ (X:Set) (p : WFT X) (q: WFT X)
    (C : X → X → Prop) (A : Prop) (B : Prop),
    SecureBy (fun y z ⇒ C y z ∨ A) p →
    SecureBy (fun y z ⇒ C y z ∨ B) q →
    SecureBy (fun y z ⇒ C y z ∨ (A ∧ B))
             (oplus_nullary p q).
  ```

- Next, we proceed one level-up, to define `oplus_unary`:

  ```
  Definition oplus_unary
    (X:Set) (p : WFT X): WFT X → WFT X.
  ```

  We’ve written the function using Coq’s tactic language because it involves a nested induction, but it’s easier to understand it operationally using the following Haskell code:

  ```
  oplus_unary q ZT = q
  oplus_unary ZT q = q
  oplus_unary p@(SUP f) q@(SUP g)
    = SUP (λx → oplus_nullary (oplus_unary (f x) q)
                              (oplus_unary p (g x)))
  ```

  There is a similar lemma about `oplus_unary`, that explains how it can be used to secure intersections of unary predicates:

  ```
  Lemma oplus_unary_sec_intersection:
    ∀ (X:Set) (p:WFT X) (q:WFT X) (C : X → X → Prop)
    (A : X → Prop) (B : X → Prop),
    SecureBy (fun y z ⇒ C y z ∨ A y) p →
    SecureBy (fun y z ⇒ C y z ∨ B y) q →
    SecureBy (fun y z ⇒ C y z ∨ (A y ∧ B y))
             (oplus_unary p q).
  ```

- Finally, we proceed yet one level up, to define `oplus_binary`:

  ```
  Definition oplus_binary
    (X:Set) (p : WFT X) : WFT X → WFT X.
  ```

  Its definition follows `oplus_unary` and it’s perhaps simpler to understand in Haskell:

  ```
  oplus_binary q ZT = q
  oplus_binary ZT q = q
  oplus_binary p@(SUP f) q@(SUP g)
    = SUP (λx → oplus_unary (oplus_binary (f x) q)
                            (oplus_binary p (g x)))
  ```

The fixpoint `oplus_binary` turns out to be exactly what we want to combine well-founded trees to secure intersections (of binary predicates). Here is the corresponding lemma:

```
Lemma oplus_binary_sec_intersection :
 ∀ (X:Set) (p : WFT X) (q : WFT X)
 (A : X → X → Prop) (B : X → X → Prop),
 SecureBy A p → SecureBy B q →
 SecureBy (fun x y ⇒ A x y ∧ B x y) (oplus_binary p q).
```

The proofs of all three lemmas above are direct and short, and their corollary is:

```
Corollary af_intersection (X:Set) (A B :X→ X→ Prop):
  almost_full A → almost_full B →
  almost_full (fun x y ⇒ A x y ∧ B x y).
```

The binary version of the Ramsey theorem is, using classical logic, a direct consequence of `af_intersection`: consider a binary relation R on `nat` and call a subset A of `nat` homogeneous iff:

- For all n and m in A such that $n < m$ it is $R\ n\ m$, *or*
- For all n and m in A such that $n < m$ it is $\neg(R\ n\ m)$.

Ramsey’s theorem states that for every binary relation R there exists an *infinite* homogeneous subset of `nat`, A. To prove this, assume by contradiction that no such infinite homogeneous subset

[3] Exercise: use Ramsey’s theorem to prove (classically) the intersection theorem, proceeding by contradiction and using a 3-coloring.

exists. This means that both R and $\neg R$ are AF, which means that their intersection is AF by `af_intersection`. But the empty relation cannot be AF because it relates no elements whatsoever!

Although we have focused on binary relations, a generalization of our development to n-ary relations (corresponding to the original version of the theorem [21]) is entirely possible. As a final remark, the intersection theorem for the case of WQOs is folklore – in the context of WQOs the transitivity assumption seems to significantly simplify the proof. For instance, such a proof is contained in a short paper by Nash-Williams [18].

4.3 Type-based combinators

In this section we show how to derive AF relations from simpler ones in a type-directed way, and how we may use them to define recursive functions.

Cofunctoriality of well-founded trees We can show that well-founded trees is a cofunctor by defining a cofmap operation

```
Fixpoint cofmap (X:Set) (Y:Set) (f:Y→ X) (p:WFT X) :=
  match p with
  | ZT ⇒ ZT Y
  | SUP w ⇒ SUP (fun y ⇒ cofmap f (w (f y)))
  end.
```

with the straightforward property, and corollary:

```
Lemma cofmap_secures:
  ∀ (X Y:Set)(f:Y→ X) (p:WFT X) (R:X→ X→ Prop),
  SecureBy R p →
  SecureBy (fun x y ⇒ R (f x) (f y)) (cofmap f p).
```

```
Corollary af_cofmap (X Y:Set) (f:Y→ X) (R:X→ X→ Prop):
  almost_full R → almost_full (fun x y ⇒ R (f x) (f y)).
```

For example, the `af_cofmap` theorem can be used when we would like to map complicated data structures to `nat` values through "ranking functions", so that we may then re-use the $\leq$ relation and the `leq_af` witness that $\leq$ is AF.

Example 2 (Use of a ranking function through cofmap)**.** Consider the following definition (in Haskell notation[4]):

```
flip1 (0,_) = 1
flip1 (_,0) = 1
flip1 (x+1,y+1) = flip1 (y+1,x)
```

Through the use of `cofmap` we may define this function by observing that the transition relation is

```
T x y := fst x <= snd y ∧ snd x < fst y
```

and taking

```
R x y := fst x + snd x <= fst y + snd y
```

as our AF relation. Showing that

```
∀ x y, clos_trans T x y ∧ R y x → False
```

is easy and the proof that `R` is AF is just (`af_cofmap leq_af`).

[4] AFExamples.v gives the Coq definition.

Finite types There is a very natural AF relation on types that have finitely many inhabitants, and that is simply the equality on elements of these types. The simplest interesting such finite type is `bool`. How would we go about constructing a tree that secures equality on `bool`? The `bool_tree` definition below provides the answer:

```
Definition bool_tree : WFT bool :=
  SUP (fun x ⇒ SUP (fun y ⇒ SUP (fun z ⇒ ZT bool))).
```

The type `bool` has two inhabitants, therefore any tree that has at least *three* uses of the `SUP` constructor secures equality! As as consequence:

```
Corollary af_bool : almost_full (@eq bool).
```

We are not going to generalize here this construction to arbitrary finite types, but the reader should be convinced that this is possible to do – a tree with $k + 1$ uses of `SUP` before returning `ZT` does the job for any finite type inhabited by k values. Our accompanying development includes this construction.

Products The intersection property and cofunctoriality are already extremely powerful – here is the simplest construction to create an AF relation for products based on these components:

```
Lemma af_product (X : Set) (Y : Set) :
  ∀ (A : X → X → Prop) (B : Y → Y → Prop),
  almost_full A → almost_full B →
  almost_full (fun x y ⇒ A (fst x) (fst y) ∧
                         B (snd x) (snd y)).
intros A B afA afB.
apply(af_intersection (@af_cofmap _ _ (@fst X Y) A afA)
                      (@af_cofmap _ _ (@snd X Y) B afB)).
Defined.
```

The proof is just applications of `af_intersection` and `af_cofmap` through the `fst` and `snd` projections out of pairs.

Of course, this is not the only AF relation on products – it's just a particular one. For instance one could completely ignore the second component of a pair

```
Lemma af_product_left (X Y : Set) (A:X→ X→ Prop) :
  almost_full A →
  almost_full (fun (x:X*Y) (y:X*Y) ⇒ A (fst x) (fst y)).
intros afA.
apply (@af_cofmap _ _ (@fst X Y) A afA).
Defined.
```

or could imagine other wild combinations through the use of cofunctoriality, unions and intersections – or as a last resort, could create something more sophisticated by hand-writing the well-founded tree witness.

We give now a small example to demonstrate the product combinator in action.

Example 3 (Lexicographic order)**.** Consider the following recursive definition (in Haskell notation):

```
flex (0,_) = 1
flex (_,0) = 1
flex (x+1,y+1) = f (x,y+2) + f (x+1,y)
```

This is an example of function where the arguments descend lexicographically. We can also observe that in any recursive call, one of the two arguments is decreasing. This immediately suggests that we should use the AF relation:

```
R x y := fxt x <= fst y ∧ snd x <= snd y
```

Recall that since $\leq$ is AF and we have already given a product combinator (`af_product`), the relation `R` is AF.

The transition relation of the program is also what you'd expect:

```
T x y := fst x < fst y ∨ (fst x = fst y ∧ snd x < snd y)
```

In the first recursive call, the first argument becomes smaller, and in the second recursive call the second argument becomes smaller, while the first remains the same. It is then quite easy to show that `∀ x y, clos_trans T x y ∧ R y x → False` and apply `af_induction` to define the recursive function.

Sums If we are given two AF relations on types `X` and `Y` respectively, is there a natural AF relation that we can define on `X+Y`? One that we find often useful is the relation that lifts these two relations in the following way:

```
Definition sum_lift (X Y:Set)
  (A:X→ X→ Prop) (B:Y→ Y→ Prop) (x:X+Y) (y:X+Y) :=
  match (x,y) with
  | (inl x0, inl y0) ⇒ A x0 y0
  | (inl x0, inr y0) ⇒ False
  | (inr x0, inl y0) ⇒ False
  | (inr x0, inr y0) ⇒ B x0 y0
  end.
```

If two elements have the same tags they are compared with one or the other relation, otherwise they are not related. We will now show that if `A` and `B` are AF, then so is `sum_lift A B`. The key intuition behind our construction is the close connection between tagged sums and products where the first component is the "tag" and the second is the value. Here is the construction step-by step.

- First, consider the following definition

  ```
  Definition left_sum_lift (X Y:Set) (A:X→ X→ Prop)
    (x:X+Y) (y:X+Y) :=
    match (x,y) with
    | (inl x0, inl y0) ⇒ A x0 y0
    | (inl x0, inr y0) ⇒ False
    | (inr x0, inl y0) ⇒ False
    | (inr x0, inr y0) ⇒ True
    end.
  ```

  which is almost like `sum_lift` but in the case of two right injections returns `True`. If we are given a well-founded tree that secures a relation `A : X → X → Prop`, we will show that the following fixpoint secures `left_sum_lift A`.

  ```
  Fixpoint left_sum_tree (X Y:Set) (p:WFT X)
    : WFT (X+Y) :=
    match p with
      | ZT ⇒ SUP (fun x ⇒ SUP (fun y ⇒
                            SUP (fun z ⇒ ZT (X+Y))))
      | SUP f ⇒ SUP (fun x ⇒
        match x with
          | inl x0 ⇒ left_sum_tree Y (f x0)
          | inr x0 ⇒ SUP (fun y ⇒
            match y with
              | inl y0 ⇒ left_sum_tree Y (f y0)
              | inr y0 ⇒ ZT (X+Y)
            end)
        end)
    end.
  ```

  What is the intuition behind this construction? If the tree is `ZT` then all elements `X` are related and hence if we take three elements in a row with `SUP` constructors we are guaranteed to secure `left_sum_lift A` either by having met two left injections, or by having met two right injections. This is exactly what we did in the case of finite types. If on the other hand the tree that secures `A` is `SUP f`, then we examine the next element `x` – if it is a left injection we may simply recurse. If `x` is a right injection, then we examine the subsequent element `y`: if `y` is a left injection we can again recurse, but if it is also a right injection then we are simply finished (since all right injections are related by `left_sum_lift A`). Formally:

  ```
  Lemma sec_left_sum_tree (X Y:Set) (p : WFT X):
    ∀ (A : X → X → Prop), SecureBy A p →
    SecureBy (left_sum_lift A) (left_sum_tree Y p).
  ```

- Our next step is to flip everything around! We will use of a `transpose` function and get symmetric versions of the previous fixpoint and lemma:

  ```
  Definition transpose (X Y:Set) (x:X+Y) : Y+X :=
    match x with
    | inl x0 ⇒ inr _ x0
    | inr x0 ⇒ inl _ x0
    end.

  Definition right_sum_lift (X Y:Set)
    (B:Y→ Y→ Prop) (x y:X+Y) :=
    left_sum_lift B (transpose x) (transpose y).

  Definition right_sum_tree (X Y:Set) (p:WFT Y)
    : WFT (X+Y) :=
    cofmap (@transpose X Y) (@left_sum_tree Y X p).

  Lemma sec_right_sum_tree (X Y:Set) (p : WFT Y):
    ∀ (B : Y → Y → Prop), SecureBy B p →
    SecureBy (right_sum_lift B) (right_sum_tree X p).
  intros. apply cofmap_secures.
  apply sec_left_sum_tree; assumption. Defined.
  ```

- We are almost there! Observe that if we are given relations `A : X → X → Prop` and `B : Y → Y → Prop` then the *intersection* of the `left_sum_lift A` and `right_sum_lift B` is precisely `sum_lift A B`. We've already shown a combinator for intersections of relations so it's now straightforward to derive our final result

  ```
  Corollary af_sum_lift (X Y : Set) :
    ∀ (A : X → X → Prop) (B : Y → Y → Prop),
    almost_full A → almost_full B →
    almost_full (sum_lift A B).
  ```

We do not present here an example of using `af_sum_lift`, but we will see examples later in Section 7.

Dependent products and recursive types We do no currently include in our development combinators for dependent products nor recursive types, though nothing seems to be prohibitive about either and we intend to extend the set of available type-based combinators in the future. The question of recursive types is of particular interest as it is a well-studied topic in the context of WQOs, where the canonical WQO for lists and more general recursive types is based on homeomorphic embeddings [18, 14]. There have been attempts to port some of these theorems for WQOs in a constructive setting (e.g. homeomorphic embeddings for lists on finite alphabets [3, 23]) so we believe this is a quite plausible direction for future work.

5. Size-change termination and AF induction

We have examined combinators on AF relations, and simple examples such as lexicographic descent. Lexicographic orders are not terribly difficult (Coq already comes with combinators to compose lexicographically two well-founded relations, in fact) but the power of the method shows itself in examples that go beyond lexicographic orders. Consider the following example.

Example 4 (Beyond lexicographic order)**.** Here is an example in Haskell-like notation:

```
gnlex (0,_) = 1
gnlex (_,0) = 1
gnlex (x+1,y+1) = gnlex (y+1,y) + gnlex (y+1,x)
```

To define this program, we will use the AF `R` for products, and the "obvious" transition relation `T`:

```
Definition T x y := (fst x = snd y ∧ snd x < snd y) ∨
                    (fst x = snd y ∧ snd x < fst y).
Definition R x y := fst x <= fst y ∧ snd x <= snd y.
```

It's now possible to show that the transitive closure of `T` has an empty intersection with (the inverse of) `R`, we have proved this lemma independently and called it `T_empty_intersect`. Using this lemma we may define `gnlex` completely as:

```
Definition gnlex: ∀ (x:nat*nat), nat.
apply af_induction with (T:=T) (R:=R).
(* prove almost_full *)
apply af_intersection;
   repeat (apply af_cofmap; apply leq_af).
(* prove intersection emptyness *)
intros x y (CT,HR).
apply (T_empty_intersect CT HR).
(* give the functional *)
refine (fun x ⇒ match x as w
        return (∀ y, T y w → nat) → nat with
        | (0,_) ⇒ fun frec ⇒ 1
        | (_,0) ⇒ fun frec ⇒ 1
        | (S x, S y) ⇒ fun frec ⇒ frec (S y, y) _ +
                                   frec (S y, x) _
              end).
unfold T in *. left. simpl; omega.
unfold T in *. right. simpl; omega.
Defined.
```

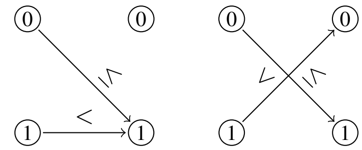
Ben-Amram [2] notices that examples like `gnlex` belong in a syntactic class of programs that can be shown terminating by *size-change termination* (SCT) [15, 12, 2] but not by a direct lexicographic descent argument, although semantically the class of *mathematical functions* one may define using size-change termination and those that can be defined with lexicographic descent orders coincide. It is then reassuring to see that examples from that class can be written quite straightforwardly!

5.1 Formal connection

In fact, the connection to size-change termination can be made more precise. The short summary of this section is that the soundness of size-change termination follows from our general `wf_from_af` lemma. For the rest of this section we show this connection, using `gnlex` as our working example.

Let us start by briefly describing the main idea behind SCT: The first step in showing that a recursive definition is terminating is to identify the various recursion patterns and abstract each as a *size-change graph*. A size-change graph for a k-argument function is a labeled graph with nodes labeled from $\{0, \ldots, k-1\}$ and arcs with labels $<$ and $\leq$.

Example 5 (Size-change graph for gnlex)**.** For our two-argument function `gnlex` we get the following two size-change graphs (arising from the transition relation `T` of the function):



G_0 G_1

A size-change graph G for a k-argument function induces a relation on k-tuples and we say that a size-change graph *approximates* a relation `T` iff $T \subseteq T_G$. In our `gnlex` example, each of the two graphs approximates a disjunct from `T`.

Size change graphs *compose* so that the composition of two arcs one of which is $<$ creates a new arc $<$, whereas the composition of two $\leq$ arcs gives a new $\leq$ arc. We write this composition with notation $G_1; G_2$ (written G_{12} for brevity below). Graph composition satisfies the following proposition.

Proposition 1. *If G_1 approximates T_1 and G_2 approximates T_2 then $G_1; G_2$ approximates $T_1 \cdot T_2$ (where $\cdot$ is transitive relation composition).*

Assume now that the transition relation of a program is given by n-disjuncts $T = T_1 \cup \ldots T_n$ each of which corresponds to some recursion pattern and is approximated by a size-change graph G_i (as in our example with $n = 2$). Size-change termination then considers the set S, defined as the transitive closure of the set $\{G_1, \ldots, G_n\}$ under graph composition.

Example 6 (Transitive closure of size-change graphs)**.** What is this set S in our `gnlex` example? If we start off with G_0 and G_1, we have to consider the compositions $G_0; G_0$, $G_0; G_1$, $G_1; G_0$, and $G_1; G_1$. We observe that G_{00} is a new graph with edges $0 \xrightarrow{<} 1$, $1 \xrightarrow{<} 1$, G_{01} is a new graph with edges $0 \xrightarrow{<} 0$, $1 \xrightarrow{<} 0$, G_{10} is *exactly* G_{00} and G_{11} is a new graph with edges $0 \xrightarrow{<} 0$, $1 \xrightarrow{<} 1$. If we continue in this fashion we can compute that the set S is just:

$$S = \{G_0, G_1, G_{00}, G_{01}, G_{11}, G_{111}\}$$

What is the importance of the set S? We have seen that T can be approximated by $\{G_0, G_1\}$ and we have seen that compositions of graphs approximate compositions of relations. This means that for every k, the composition of T with itself k times T^k (which we will call the k-th *power* of T) can be approximated by the set of graphs in S (which will typically, as in our example, be finite): Precisely, for every x and y such that $T^k\ x\ y$ it is the case that $T_G\ x\ y$ for some $G \in S$. This is wonderful news, because it enables the following lemma.

Lemma 1. *Assume that $T = T_1 \cup \ldots \cup T_n$ and G_i approximates T_i, and let S be the transitive closure of the set $\{G_i, \ldots, G_n\}$. If every $G \in S$ induces a relation T_G such that $T_G \cap R^{-1} = \emptyset$ for some AF R then T is well-founded.*

Proof. By `wf_from_af` we only have to show that for all x and y such that $T^+ x\ y$ it is not the case that $R\ y\ x$. If $T^+ x\ y$ then there exists some k such that $T^k x\ y$, which means that there exists some $G \in S$ such that $T_G\ x\ y$ and we know that $T_G \cap R^{-1} = \emptyset$. □

Next, consider the size-change graph I with edges $i \xrightarrow{\leq} i$ for each i, and let us call the induced relation $T_I\ x\ y = \bigwedge x_i \leq y_i$. By the constructive Ramsey Theorem, `af_intersection`, T_I is AF.

Example 7. We can now show that `gnlex` is terminating by checking that every graph $G \in S$ has empty intersection with $(T_I)^{-1}$ and using Lemma 1.

Size-change termination uses the same AF relation T_I and Lemma 1, through the following auxiliary lemma.

Lemma 2. *If G approximates T and some power G^n of G contains an arc $i \xrightarrow{<} i$ then $T \cap T_I^{-1} = \emptyset$.*

Proof. Assume that $T\ x\ y$ and $T_I\ y\ x$. We then have $(T{\cdot}T_I)\ x\ x$ and $(T{\cdot}T_I)^n\ x\ x$. But I approximates T_I and because compositions of graphs approximate compositions of relations and $G; I = G$ it follows that G^n approximates $(T{\cdot}T_I)^n$. this means that $x_i < x_i$, which is a contradiction.[5] ☐

We are now ready to state and prove the basic SCT principle.

Theorem 1 (Size-change termination)**.** *Assume that $T = T_1 \cup \ldots \cup T_n$ and G_i approximates T_i, and let S be the transitive closure of the set $\{G_i, \ldots, G_n\}$. If every $G \in S$ has a power with an arc $i \xrightarrow{<} i$ then T is well-founded.*

Proof. By Lemma 2 we know that $T_G \cap T_I^{-1} = \emptyset$ for every $G \in S$, and by Lemma 1 we are done. ☐

Hence, we have proved the size-change termination condition relying on our `wf_from_af` theorem. The reader can observe that the condition is true for the set S we have computed for `gnlex`.

As a side-note, sometimes the size-change termination criterion is stated by requiring that every idempotent graph $G \in S$ has an arc $i \xrightarrow{<} i$, which is an equivalent condition since any size-change graph has an idempotent power.

We have not formalized this connection in Coq, but this formalization seems like an interesting direction for future work, especially combined with tactics to extract automatically size-change graphs from Coq recursive definitions. For now we will simply state that SCT can be proved using our `wf_from_af` and leave the development of an SCT-based tactic as future work. Finally, the literature on SCT is also concerned with *mutual induction*. We show how to define mutually inductive fixpoints using AF relations in Section 7.

6. The Terminator rule

We have used online termination and WQOs as a way to approach AF relations and `af_induction`, but it turns out that `af_induction` is general enough to capture the proof principle behind Terminator [7, 20]. The key theorem behind Terminator is the *disjunctive well-foundedness* statement below:

> If $R_1 \ldots R_n$ are well-founded for some finite n, and $R^+ \subseteq R_1 \cup \ldots \cup R_n$ then R is well-founded.

The proof of this theorem relies on a Ramsey argument [20], but here we will simply prove it – intuitionistically – by using theorem `wf_from_af` from Section 3.1. Here it is, together with the proof:

```
Lemma disjunctive_wf :
  ∀ (A:Set) (T : A → A → Prop)
  (R1 R2 : A → A → Prop)
  (decR1 : dec_rel R1) (decR2 : dec_rel R2),
  well_founded R1 → well_founded R2 →
  (∀ x y, clos_trans_1n A T x y → R1 x y ∨ R2 x y) →
  well_founded T.
intros A T R1 R2 decR1 decR2 wfR1 wfR2 Hincl.
pose (R x y := not (R1 y x) ∧ not (R2 y x)).
assert (almost_full R) as Raf.
  apply af_intersection.
  apply (af_from_wf wfR1 decR1).
  apply (af_from_wf wfR2 decR2).
destruct Raf as (p,Hsec).
apply wf_from_af with (R:=R) (p:=p).
intros x y CT. destruct CT; firstorder. assumption.
Defined.
```

[5] The argument is still constructive as we can represent $T \cap T_I^{-1} = \emptyset$ as $\forall xy, T\ x\ y \wedge T_I\ y\ x \rightarrow$ `False`.

For the sake of demonstration, we have stated and proved the theorem when $n = 2$ but a generalization is trivial. The proof is instructive as well: we do it by appealing to `wf_from_af` (Section 3.1), and instantiating:

```
R := fun x y ⇒ not (R1 y x) ∧ not (R2 y x)
```

We next show that `R` is AF by using our constructive Ramsey `af_intersection` lemma and `af_from_wf` (Section 3). Finally the intersection emptyness condition is trivial to show.

We can easily then use disjunctive well-foundedness to deduce the standard Terminator proof rule (for the union of two WF relations):

```
Lemma disj_wf_induction:
 ∀ (A:Set) (T : A → A → Prop)
 (R1 R2 : A → A → Prop)
 (decR1 : dec_rel R1) (decR2 : dec_rel R2),
 well_founded R1 → well_founded R2 →
 (∀ x y, clos_trans_1n A T x y → R1 x y ∨ R2 x y) →
 ∀ P : A → Set,
 (∀ x, (∀ y, T y x → P y) → P x)
 → ∀ a, P a.
```

7. Discussion

We continue with several points that deserve further discussion or highlight directions for future work.

Power-induction Recall that, given a transition relation `T` and an AF relation `R`, our proof obligation for `af_induction` is:

```
∀ x y, clos_trans_1n X T x y ∧ R y x → False
```

Often, it might be tedious to find the right generalization of this statement to prove it inductively on the transitive closure. For this reason we include a powerful generalization of our induction principle, which we call *power-induction*:

```
Lemma af_power_induction:
  ∀ (A:Set) k
  (T : A → A → Prop) (R : A → A → Prop),
  k >= 1 → almost_full R →
  (∀ x y,
    clos_trans_1n A (power k T) x y ∧ R y x → False) →
  ∀ P : A → Set,
  (∀ x, (∀ y, T y x → P y) → P x) →
  ∀ x, P x.
```

We use notation `power k T` for the k-th transitive composition of `T` with itself (written earlier as T^k). For instance,

```
power 1 T = fun x y ⇒ T x y
power 2 T = fun x y ⇒ ∃ z, T x z ∧ T z y
```

and so on. The power of `af_power_induction` is that it allows for transition relations that don't immediately exhibit some argument metric going down, but they do so after some (k) recursive calls. This is akin to *inlining* a recursive definition.

Here is another example that can be programmed nicely using power-induction and the sum combinators we've presented earlier.

Example 8 (Combining power-induction and sums)**.** In Haskell-like notation:

```
fsum (inl 0)         = 1
fsum (inl (S x))     = fsum (inr (x+2))
fsum (inr x) | x < 2 = 0
fsum (inr x)         = fsum (inl (x-2))
```

The interesting observation in this example is that, starting from a left injection, after two recursive calls we are back at a left injection and the value has decreased. Similarly for a right injection. Let us use this intuition and define the AF relation `Rfsum`, which uses our sum combinator `sum_lift`, along with the (obvious) transition relation `Tfsum` below:

```
Definition Tfsum := fun x y ⇒
   (∃ x0, x = inr nat (x0+2) ∧ y = inl nat (S x0)) ∨
   (∃ x0, x = inl nat (x0-2) ∧ y = inr nat x0).
Definition Rfsum := sum_lift le le.
```

Notice that `(power 2 T)` satisties a nicer invariant (under transitive composition) than `T` and we may use AF induction for the `(power 2 T)` relation:

```
Definition fsum: ∀ (x:nat+nat), nat.
apply af_power_induction
  with (T := Tfsum) (R := Rfsum) (k := 2). omega.
(* prove almost_full *)
apply af_sum_lift. apply leq_af. apply leq_af.
(* prove intersection emptyness *)
apply fsum_pow2_empty.
(* give the functional *)
refine (fun x ⇒ match x as w
        return (∀ y, Tfsum y w → nat) → nat with
        | inl 0 ⇒ fun frec ⇒ 1
        | inl (S x) ⇒ fun frec ⇒ frec (inr nat (x+2)) _
        | inr x ⇒ fun frec ⇒ frec (inl nat (x-2)) _
        end).
left. ∃x0. intuition.
right. ∃x0. intuition.
Defined.
```

Notice that we use $k = 2$ and we rely on an independently proven lemma about the intersection emptyness, `fsum_pow2_empty`. Actually, we can repeat the same definition by using ordinary AF induction ($k = 1$) but doing so requires us to prove a more complicated invariant about `Tfsum`. Happily, choosing $k = 2$ makes the proof of the intersection emptyness much simpler.

Mutual induction We can easily derive mutual induction principles using AF induction, and we outline here the basic idea for two mutually recursive functions – the accompanying development gives the full details.

Suppose that we want to define two mutually recursive functions `f : A → C` and `g : B → D`. Suppose that the transition relation for calls from `f` to itself is `TAA:A→ A→ Prop` and for calls from `f` to `g` is `TBA:B→ A→ Prop`. Similarly calls from `g` to itself are described by `TBB:B→ B→ Prop` and calls from `g` to `f` are described by `TAB:A→ B→ Prop`. We may consider the sum `A + B` and "lift" the relations `TAA`, `TBA`, `TAB`, `TBB` with `lift_rel_union TAA TBB TAB TBA` which operates on `A + B` in the following way: if both arguments are left injections use `TAA`, if they are both right injections use `TBB`; otherwise use `TAB` and `TBA` for each case.

If we are given two almost-full relations `RA:A→ A→ Prop` and `RB:B→ B→ Prop` then we can show that the mutually recursive definition is well-formed when

```
∀ x y, clos_trans_1n (A+B)
             (lift_rel_union TAA TBB TAB TBA) x y ∧
             sum_lift RA RB y x → False
```

Recall that `sum_lift` returns `False` if the tags do not match, otherwise compares the arguments using `RA` or `RB`. Intuitively, we tag each argument with the "name" of the function it is passed to, and we require that, whenever we return to an argument with the same tag (transitively) the intersection with an AF relation be empty.

Our development derives such a mutual induction principle (but slightly more general), called `af_mut_induction` and shows how one can use it to define fixpoints like:

```
f 0     = 1
f (x+1) = f x + g (x+2)
g x | x < 2 = 1
g (x+2)     = f x
```

The interesting bit in this example is that the argument $x + 1$ in the first recursive call to g does not decrease – in fact it increases and only when we return to a call to f through g does it decrease.

The design of convenient and general mutual induction principles is a topic that is subject to many engineering decisions and seems an excellent direction for future work. For instance, we could define very easily a k-ary mutual induction principle on k functions that all accept arguments of type `A` by using a similar methodology: lift the transition relations to type `(A * Finite k)` using the finite types to tag arguments with function identifiers, and lift an AF relation on `A` to `(A * Finite k)` using the `af_intersection` theorem and the fact that equality on finite types is an AF relation.

The computational content of the proofs Our development is constructive, so one might wonder about the computational content of our proofs. Recall corollary `af_inf_chain` from Section 2.1:

```
Corollary af_inf_chain (X : Set) (R : X → X → Prop):
   almost_full R →
   ∀ (f : nat → X), ∃n, ∃m, (n > m) ∧ R (f m) (f n).
```

In fact we can write a function that, given a `WFT X` tree `p`, and an infinite sequence `f : nat → X` computes the length of a prefix of the sequence in which there exist two related elements:

```
Fixpoint aux_length X (p:WFT X) (f:nat→ X) (k:nat) :=
  match p with
  | ZT ⇒ k
  | SUP g ⇒ aux_length (g (f k) f (S k)).
  end.
Definition length X (p : WFT X) f := aux_length p f 0
```

In `aux_length`, the variable `k` is the cursor into the sequence `f`, as in the proof of `sec_binary_infinite_chain` in Section 2.1.

One expects the `length` function to terminate immediately when `f` is the trivial sequence $0, 0, \ldots$ `(fun (x:nat) ⇒ 0)` and `p` is the well-founded tree that secures $\leq$ (let us call this `leq_wft`). Indeed that is the case, since our construction of the well-founded tree for $\leq$ is based on comparing two consecutive elements for $<$.

Surprisingly, if `p = oplus_binary leq_wft leq_wft` (which secures the same relation, $\leq$) then the call to `length p (fun x ⇒ 0)` does not terminate almost immediately, does not slow down as computation continues, and does not consume increasingly more memory (in Coq or Haskell, where we've also implemented this for comparison): it seems to loop!

Of course it doesn't *actually* loop – the reason for this behavior is the exponential nature of the combinators to form trees for intersections of relations. Notice that `length p (fun x ⇒ 0)` gives the length of the left-most path in the well-founded tree `p` and we can compute the size of this left-most path following the structure of `oplus_nullary`, `oplus_unary`, and `oplus_binary`:

```
lm_nullary x y = x + y

lm_unary 0 y = y
lm_unary x 0 = x
lm_unary (x+1) (y+1) = 1+lm_nullary (lm_unary x (y+1))
                                    (lm_unary (x+1) y)
lm_binary 0 y = y
lm_binary y 0 = y
lm_binary (x+1) (y+1) = 1+lm_unary (lm_binary x (y+1))
                                   (lm_binary (x+1) y)
```

The leftmost path of `leq_wft` is just 2 and hence we are interested in `lm_binary 2 2`, which gives us the enormous bound 125412590536309936861848O!

This discussion suggests that our combinators for composing AF relations cannot be directly used to replace or improve history-based online termination testing. For instance, the resulting termination test for the sequence $(0, 0), \ldots$ (which is based on intersection of relations $\leq$ for the first and second components of a pair) would consume very little memory but would reject the sequence after 1254125905363099368618480 elements in the sequence!

It seems an interesting direction to explore whether there exists a more "efficient" version of these combinators. On the other hand, for type theory (or Coq) these enormous bounds do not appear to cause any problems at all.

Prop versus Set witnesses The previous discussion about the computational content of our proofs raises another question. Why did we separate well-founded trees from the `SecureBy` predicate, and didn't we simply define a single inductive predicate for both:

```
Inductive AF X : (X → X → Prop) : Prop :=
 | AF_ZT :∀ R, (∀ x y, R x y) → AF R
 | AF_SUP :
     ∀ R, (∀ x, AF (fun y z ⇒ R y z ∨ R x y)) → AF R.
```

There is no significant obstacle associated with this definition but we have not carried out the experiment – following a similar stratification in previous work. Actually, it may be advantageous for code extraction to have arguments that only live in `Prop`. On the other hand, having to deal with concrete `Set`-based witnesses (`WFT X`) felt reassuring and made porting some of this development to Haskell quite straightforward when investigating the computational content of the AF combinators.

8. Related work

We have already seen the proof principle behind Terminator [7, 20] in Section 6. But why has that proof principle been so successful? One answer is *composability*: The way the tool works in practice is by rewriting the program to capture the transitive closure of the transition relation R using some new program variables and then try to synthesize well-founded relations $R_1 \ldots R_n$ so that $R^+ \subseteq R_1 \cup \ldots \cup R_n$ from static analysis of the code. The way this synthesis works is by starting off with the empty union, for which $R^+ \not\subseteq \emptyset$, a fact that can be used to derive a well-founded relation R_1. Next, a similar check is made that $R^+ \subseteq R_1$. If this test fails, Terminator uses the failed proof to discover yet another well-founded relation R_2 and this time check $R^+ \subseteq R_1 \cup R_2$. The process continues until a termination argument is found. The key to the success of this method has been that the termination test simply uses unions of well-founded relations (instead of trying to discover more complex ways to compose them) and throws the "hard" part of the proof of checking that $(R^+ \subseteq R_1 \cup \ldots \cup R_n)$ to extremely powerful external tools such as SMT solvers. Interestingly, Terminator performs something like AF-power-induction if the termination arguments fail for the transition relation: it starts unrolling loops until termination argument synthesis is able to find an answer.

Porting some of Ramsey theory in a constructive setting seems to have been a fascinating subject among mathematicians and computer scientists, since the original proof and definitions seem hopelessly classical. Our development is based on Veldman's original ideas [28, 9]. This is not to say that our development is the only possible way to develop constructive Ramsey-like arguments, for instance there exists an alternative formulation [8] but does not seem as suitable for termination purposes as the one we present in this paper. Similarly, constructive proofs of various homeomorphic embedding lemmas (such as Higman's Lemma [3, 23, 10]) have appeared in the literature. Our development seems to be the first that connects constructive Ramsey theory and termination proving.

Nowadays there exists a large set of recursion-encoding techniques in type theory and Coq, some of which include good support for automation. The most straightforward way to program recursion in Coq [4] is either by structural recursion or by using subset types [26] and `measure` arguments. An extension of "guarded" recursion (and co-recursion) implemented in a variant of Agda is sized-types [1] (not to be confused with size-change termination).

The Bove and Capretta method [5] is traditionally the *de-facto* way to define recursive programs that include complex argument relations in Type Theory: For each recursive definition the user introduces an indexed type family with each constructor corresponding to a particular recursion pattern and indices that correspond to the program variables. The function can then be defined by induction on this witness and dependent pattern matching. After-the-fact, the programmer can provide such an inductive witness at the call-sites, to justify the totality of their definitions. Kraus *et al.* [13] propose a related technique for showing automatically the termination of Isabelle functions by extracting their *inductive graph* and using an induction principle on that graph (that graph roughly corresponds to the Bove-Capretta inductive witness).

Charguéraud [6] presented recently a fixpoint combinator which uses, internally, extra measure arguments but hides them from the programmer. Charguéraud, in an impressive development, uses previous work on recursion theory (the work on "optimal fixpoints") to fully separate the definition of a recursive function from the termination argument. The result is a beautifully engineered library that has been used to show many difficult examples from previous work. In spirit, Charguéraud's technique is closer to well-founded relations and measures than AF relations, and it'd be interesting to explore whether some of his ideas for the separation of code and termination arguments can be reused in our development.

Focusing on practicality, Megacz [17] presents an extremely pleasant to the user monadic way to structure recursive definitions based on a coinductive datatype, which allows one to prove that the recursive definition will terminate after-the-fact. To our surprise, Megacz' method is rarely cited in related work.

9. Directions for future work

We have already discussed several possibilities for future work in Sections 5 and 7. Some important directions are the design of more induction principles, such as more general and convenient versions of AF mutual induction principles, and ways to bring our development closer to size-change termination.

Although we have identified an appealing and relatively unexplored induction principle, further investigation is required to make our approach more practical. Related work has identified several requirements that have to be met before a termination library or methodology can be deemed effective or successful. It has to: (i) allow one to define complex recursion patterns naturally (preferrably by giving code but deferring proof obligations), (ii) allow computation with recursive definitions, (iii) provide unfolding theorems that unfold a recursive function at will, and (iv) provide the ability to reason about recursive functions using induction. In the context of a programming language that requires totality checking, the first two requirements matter the most but for a proof assistant the latter two are equally important and we plan to investigate these reasoning principles in future work.

Another ambitious direction is tool support and automation, as well as the integration of our framework in a practical dependently typed language or proof assistant. The biggest challenge there is to develop a methodology so that a tool (be it an interactive environment, or an type checker) may give feedback to the programmer to help him synthesize the right termination argument, or automatically discharge the various relation inclusion obligations. We are optimistic about this direction, because of (i) the tremendous recent progress in SMT solvers and automated reachability checking and (ii) the composability of AF relations. Research languages such as Trellys[6] and Agda [19] could potentially provide good candidates for this kind of tool integration.

[6] `http://code.google.com/p/trellys/`

References

[1] Andreas Abel. Termination and guardedness checking with continuous types. In M. Hofmann, editor, *Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain*, volume 2701 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2003.

[2] Amir M. Ben-Amram. General size-change termination and lexicographic descent. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, volume 2566 of Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 2002.

[3] Stefan Berghofer. A constructive proof of Higman's Lemma in Isabelle. In *TYPES*, pages 66–82, 2003.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[5] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical. Structures in Comp. Sci.*, 15:671–708, August 2005.

[6] Arthur Charguéraud. The optimal fixed point combinator. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceeding of the first international conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 195–210. Springer, 2010.

[7] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[8] Thierry Coquand. An analysis of Ramsey's Theorem. *Inf. Comput.*, 110:297–304, May 1994.

[9] Thierry Coquand. A direct proof of Ramsey's Theorem. Unpublished note, available from `http://www.chalmers.se/~coquand/intuitionism.html`, 2010.

[10] Daniel Fridlender. Higman's Lemma in Type Theory. In *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, pages 112–133. Springer-Verlag, 1997.

[11] Matthias Heizmann, Neil D. Jones, and Andreas Podelski. Size-change termination and transition invariants. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 22–50, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Neil D. Jones and Nina Bohr. Call-by-value termination in the untyped lambda-calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[13] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl. Termination of Isabelle functions via termination of rewriting. In *Proceeding of the second international conference on Interactive Theorem Proving (ITP)*, 2011.

[14] JB Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc*, 95:210–225, 1960.

[15] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. ACM.

[16] Michael Leuschel. The essence of computation. chapter Homeomorphic embedding for online termination of symbolic methods, pages 379–403. Springer-Verlag, New York, NY, USA, 2002.

[17] Adam Megacz. A coinductive monad for Prop-bounded recursion. In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007*, pages 11–20, New York, NY, USA, 2007. ACM.

[18] Crispin S.J.A. Nash-Williams. On well-quasi-ordering finite trees. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 59, pages 833–835. Cambridge Univ Press, 1963.

[19] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[20] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[21] F. P. Ramsey. On a problem of formal logic. *Proceedings of The London Mathematical Society*, s2-30:264–286, 1930.

[22] F. Richman and G. Stolzenberg. Well-quasi-ordered sets. *Advanced Mathematics*, pages 145–193, 1993.

[23] Monika Seisenberger. An inductive version of Nash-Williams' minimal-bad-sequence argument for Higman's Lemma. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, pages 233–242, London, UK, 2002. Springer-Verlag.

[24] Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 71–84, New York, NY, USA, 2007. ACM.

[25] Morten H. Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

[26] Matthieu Sozeau. Subset coercions in Coq. In *In Selected papers from the International Workshop on Types for Proofs and Programs (TYPES06*, pages 237–252. Springer, 2006.

[27] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.

[28] Wim Veldman and Marc Bezem. Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics. *Journal of the London Mathematical Society*, s2-47(2):193–211, 1993.