Termination Combinators Forever

Maximilian Bolingbroke
University of Cambridge
mb566@cam.ac.uk

Simon Peyton Jones   Dimitrios Vytiniotis
Microsoft Research Cambridge
{simonpj,dimitris}@microsoft.com

Abstract

We describe a library-based approach to constructing termination tests suitable for controlling termination of symbolic methods such as partial evaluation, supercompilation and theorem proving. With our combinators, all termination tests are correct by construction. We show how the library can be designed to embody various optimisations of the termination tests, which the user of the library takes advantage of entirely transparently.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages

General Terms Algorithms, Theory

1. Introduction

The question of termination arises over and over again when building compilers, theorem provers, or program analysers. For example, a compiler may inline a recursive function once, or twice, but should not do so forever. One way to extract the essence of the problem is this:

> **The online termination problem.** Given a finite or infinite sequence of terms (often syntax trees), $x_0, x_1, x_2, ...$, with the elements presented one by one, shout "stop" if the sequence looks as if it is diverging. Try not to shout "stop" for any finite sequence; but guarantee to shout "stop" at some point in every infinite sequence.

The test is "online" in the sense that the terms are presented one by one, and the entity producing the terms is a black box. In contrast, static, offline termination checkers analyse the producing entity and try to prove that it will never generate an infinite sequence.

Termination is a well-studied problem (Section 8) and many termination tests are known. But building *good* online termination tests is hard. A good test is

- *Sound*: every infinite sequence is caught by the test.
- *Lenient*: it does not prematurely terminate a sequence that is actually finite. As an extreme example, shouting "stop" immediately is sound, but not very lenient.
- *Vigilant*: sequences of terms that are clearly growing in an "uninteresting" way are quickly reported as such — the termination test "wait for a million items then say stop" is not what we want.

Haskell'11, September 22, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0860-1/11/09…$10.00

These properties are in direct conflict: making a test more lenient risks making it less vigilant, or indeed unsound. Termination tests are typically tailored for a particular application, and it is all too easy to inadvertently build tests that are either unsound or too conservative.

Our contribution is to describe how to *encapsulate termination tests in a library*. Termination tests built using our library are guaranteed sound, and the library embodies standard (but tricky) techniques that support leniency and vigilance. Our specific contributions are these:

- We give the API of a combinator library that allows the client to construct sound, lenient, and vigilant termination tests (Section 2). Our API is *modular* and *compositional*: that is, you can build complex tests by combining simpler ones.
- An API is not much good unless you can implement it. Building on classical work we show how to implement a termination test in terms of a so-called *well-quasi-order* (WQO) on the underlying type (Section 3). WQOs compose well, and we give combinators for sums, products, finite maps, and so on.
- Termination tests for recursive types are particularly interesting (Section 5). We generalise the classic homeomorphic embedding to our setting, and show what proof obligations arise.
- We show that some useful improvements to termination tests can be incorporated, once and for all, in our library: Section 6.
- We show that our library subsumes several well-studied termination tests, including homeomorphic embedding [1], and tag-bags [2] (Section 7). We further show how our combinators can capture a novel and slightly stronger version of the tag-bag termination test (Section 7.4).

To our knowledge, this is the first time that anyone has even identified an online termination tester as a separable abstraction, let alone provided a library to let you build such a thing. Yet an online termination-testing heuristic is built into the guts of many symbolic programs, including compilers (don't inline recursive functions forever) and theorem provers (don't explore unproductive proofs forever). We do not claim that our termination testers are better than any particular competing ones; rather, our library is a domain-specific language that makes it easy to explore a rich variety of online termination testers, while still guaranteeing that each is sound.

2. The client's eye view: tests and histories

Our goal is to define termination tests over terms whose type is under the control of the user. Recall that the client produces successive terms $x_0, x_1, x_2...$, and the business of the termination test is to shout "stop" if the sequence looks as if it might diverge. (In the literature the term "blow the whistle" is often used instead of "shout stop".) A possible API is thus:

```
data TTest a   -- Abstract
testSequence :: TTest a → [a] → Bool
```

Here a $TTest\ A$ is a termination tester for type A. If we have such a tester, we can test a sequence of values of type $[A]$ using the function $testSequence$; a result of $True$ means that the tester shouts "stop".

We will return to the question of construction of $TTest$ values shortly, but we can already see one problem with the API. The client produces values $x_0, x_1, ...$, one a time. As each value is produced we want to ask "should we stop now". We can certainly do this with $testSequence$, by calling it on arguments $[x_0]$, $[x_0, x_1]$, $[x_0, x_1, x_2]$, and so on, but it could be terribly inefficient to do so. Each call to $testSequence$ may have to check the entire sequence in case earlier elements have changed, rather than just looking at the most recent addition. A better API would allow you to say "here is one new value to add to the ones you already have". Thus:

```
data History a   -- Abstract
initHistory :: TTest a → History a
test :: History a → a → TestResult a
data TestResult a = Stop | Continue (History a)
```

A $History\ A$ is an abstract type that embodies the knowledge about the terms (of type A) seen so far. The function $initHistory$ creates an empty history from a termination test. Given such a history, a client can use $test$ to extend the history with one new term. The $test$ indicates that it has blown the whistle by returning $Stop$. Otherwise it returns a new history, augmented with the new term.

That leaves the question of how one creates a termination test in the first place. The exact test you want to use will depend greatly on the application, and so it is crucial that there is significant flexibility in defining them. Our library is therefore structured as a number of composable combinators to allow flexibility and rapid experimentation.

Our combinator library uses a type directed approach. A subset of the API is as follows:

```
intT    :: TTest Int
boolT   :: TTest Bool
pairT   :: TTest a → TTest b → TTest (a, b)
eitherT :: TTest a → TTest b → TTest (Either a b)
cofmap  :: (a → b) → TTest b → TTest a
```

We provide built-in tests for Int and $Bool$, and a way to compose simple tests together to make more complex ones. (We will tackle the question of recursive types in Section 5.)

Note that $TTest$ is abstract, so that the client can only construct termination tests using the combinators of the library. That is the basis for our guarantee that the termination test is sound.

As an example, here is how a client could make a $History$ that (via $test$) can be use to monitor sequences of $(Int, Bool)$ pairs:

```
myHistory :: History (Int, Bool)
myHistory = initHistory (intT `pairT` boolT)
```

An artificial example of how this $History$ could be used to implement an online termination test follows. Let's say that we have a possibly-infinite list $vals :: [(Int, Bool)]$ from which we would like to take the last item. However, the list is potentially infinite, and we would like to give up and return an intermediate element if we don't reach the end of the list promptly. A suitable value $vals\_last$ can be obtained as follows:

```
vals_last :: (Int, Bool)
vals_last = go myHistory init_lst init_rst
  where
    (init_lst : init_rst) = vals
    go hist lst rst = case test hist lst of
      Continue hist′ | (lst′ : rst′) ← rst
        → go hist′ lst′ rst′
      _ → lst
```

We know that $vals\_last$ will be defined (if the elements of $vals$ are, and $vals$ has at least one item in it) because the termination test promises to eventually shout "stop". As long as our termination test $intT$ ‘$pairT$‘ $boolT$ is reasonably lenient we can expect to extract the final value from a truly finite $vals$ list with high probability, while still gracefully failing for "bad" infinite lists.

More realistic (but more complicated) examples can be found in Section 7.

3. Termination tests and well-quasi-orders

Now that we have sketched the API for our library, we turn to the question of implementing it. The way that humans intuitively look for termination is to find a totally-ordered, well-founded "measure" and check that it is decreasing. For example, if each member of a sequence of syntax trees has strictly fewer nodes than the preceding member, the sequence cannot be infinite; here the measure is the number of nodes in the tree.

The trouble is that it can be difficult to find a simple, strictly-decreasing measure, except ones that are absurdly conservative, especially when the elements are trees. For example, the size-reduction criterion on syntax trees is sound, but far too conservative: in a compiler, inlining a function often increases the size of the syntax tree, even though progress is being made.

This is a well-studied problem [3]. The most widely-used approach is to use a so-called *well-quasi-order* (WQO) instead of a well-founded order. In this section we'll explore what WQOs are, why they are good for termination testing, and how to build WQOs using our library.

3.1 What is a WQO?

Definition 3.1. A *well-quasi-order* on A is a transitive binary relation $\trianglelefteq \in A \times A$, such that for any infinite sequence $\overline{x}^\infty \in A^\infty$, there exist $i, j > i$ such that $x_i \trianglelefteq x_j$.

For example $\leq$ is a WQO on the natural numbers; in any infinite sequence of natural numbers there must be an x_i, x_j with $i < j$, and $x_i \leq x_j$. However, a WQO $\trianglelefteq$ is not *total*; that is, there may be pairs of elements of A that are not related by $\trianglelefteq$ in either direction. A WQO is transitive by definition, and is necessarily reflexive:

Lemma 3.1. *All well-quasi-orders are reflexive.*

Proof. For any $x \in A$, form the infinite sequence $x, x, x, \ldots$. By the well-quasi-order property it immediately follows that $x \trianglelefteq x$. □

The significance of a WQO is that every infinite sequence has at least one pair related by the WQO. (In fact, infinitely many such pairs, since the sequence remains infinite if you delete the pair thus identified.) We say that a sequence $\overline{x}$ is *rejected by* $\trianglelefteq$ if there exists such a pair:

Definition 3.2. A finite or infinite sequence $\overline{x} \in \overline{A}$ is *rejected by relation* R if $\exists i, j > i.\ R(x_i, x_j)$. A sequence is *accepted* if it is not rejected.

The relation $\trianglelefteq$ is a WQO if and only if every infinite sequence is rejected by $\trianglelefteq$[1]. Hence, given an implementation of $TTest$ that uses WQOs, it is easy to implement a $History$:

```
data TTest a = WQO {(⊴) :: a → a → Bool}
newtype History a = H {test :: a → TestResult a}
```

[1] In the literature, a sequence is "good for $\trianglelefteq$" iff it is rejected by $\trianglelefteq$. This terminology seems back to front in our application, so we do not use it.

```
initHistory :: ∀ a. TTest a → History a
initHistory (WQO (⊴)) = H (go [ ])
where
  go :: [ a ] → a → TestResult a
  go xs x
    | any (⊴ x) xs = Stop
    | otherwise     = Continue (H (go (x : xs)))
```

A termination test, of type $TTest$, is represented simply by a WQO. A $History$ closes over both the WQO $\trianglelefteq$ and a list xs of all the values seen so far. The invariant is that xs is accepted by $\trianglelefteq$. When testing a new value, we compare it with all values in xs; if any are related to it by wqo, we blow the whistle by returning $Stop$; otherwise we extend xs and $Continue$.

Notice that basing a termination test on a WQO is somewhat less efficient than basing it on a total, well-founded measure, because in the latter case we could maintain a single monotonically-decreasing value, and blow the whistle if the newly presented value is not smaller. In exchange WQOs are simpler, more composable, and more lenient. In Section 6.1, we will show how we can use the fact that well-quasi-orders are transitive to reduce the length of history, which would otherwise get extended by one element each and every time $test$ is called.

3.2 Why WQOs are good for termination tests

WQOs make it easier to construct good termination tests. For example, suppose we are interested in termination of a sequence of finite strings, consisting only of the 26 lower-case letters; for example

$$\begin{array}{ll} [\,abc, ac, a\,] & (1) \\ [\,a, b, c\,] & (2) \\ [\,c, b, a\,] & (3) \\ [\,aa, ccc, bbbbaa, ca\,] & (4) \end{array}$$

One can invent a total order on such strings, based on their length, or on their lexicographic ordering, but it is not altogether easy to think of one for which all the above sequences are strictly decreasing.

Here is a WQO on such strings, inspired by Mitchell [2]:

$$s_1 \trianglelefteq_s s_2 \quad \text{iff} \quad set(s_1) = set(s_2) \text{ and } \#s_1 \leq \#s_2$$

where $set(s)$ is the set of characters mentioned in s, and $\#s$ is the length of s. Notice that strings for which $set(s_1) \neq set(s_2)$ are unrelated by $\trianglelefteq_s$, which makes it harder for $\trianglelefteq_s$ to hold, and hence makes the corresponding termination test more lenient. For example, all the sequences (1-4) above are good for this WQO.

But is this relation really a WQO? The reader is invited to pause for a moment, to prove that it is. Doing so is not immediate – which is a very good reason for encapsulating such proofs in a library and do them once rather than repeatedly for each application. Anyway, here is a proof:

Theorem 3.2. *The relation $\trianglelefteq_s$ is a well-quasi-order.*

Proof. Transitivity of $\trianglelefteq_s$ is straightforward, but we must also check that every infinite sequence is rejected by $\trianglelefteq_s$. Suppose we have an infinite sequence of strings. Partition the sequence into at most 2^{26} sub-sequences by set equality. At least one of these sequences must also be infinite, say $\overline{x}^\infty$. The length of the strings in this sequence cannot be strictly decreasing (since lengths are bounded below by zero). So we can find two elements x_i, x_j with $i < j$ and $x_i \trianglelefteq_s x_j$. □

It is often useful to find a relation that is as sparse as possible, while still remaining a WQO. For example, when solving the online termination problem we wish to delay signalling possible divergence for as long as we reasonably can.

Following this principle, me can make our string example sparser still like this:

$$\begin{array}{lll} s_1 \trianglelefteq_t s_2 & \text{iff} & set(s_1) = set(s_2) \text{ and} \\ & & \forall c \in [a...z].\ N(s_1, c) \leq N(s_2, c) \end{array}$$

where $N(s, c)$ is the number of occurrences of letter c in string s. So $s_1 \trianglelefteq_t s_2$ only if s_1 has no more a's than s_2, *and* no more b's, *and* no more c's, etc. These conjunctions make it even harder for $s_1 \trianglelefteq_t s_2$ to hold. Exercise: prove that this too is a WQO.

We can quantify how lenient a WQO is by asking how long a sequence it can tolerate. One measure of lenience is something we call the *characteristic index*.

Definition 3.3 (Characteristic index)**.** The characteristic index $K(\trianglelefteq, \overline{x}^\infty)$ of a WQO $\trianglelefteq$, relative to a finite or infinite sequence $\overline{x}^\infty$, is the largest index n for which $x_0, \ldots, x_n$ is accepted by $\trianglelefteq$.

One WQO is (strictly) more lenient than another if it always has a bigger characteristic index:

Definition 3.4 (Lenience)**.** A WQO $\trianglelefteq_1$ is more lenient than $\trianglelefteq_2$ if $K(\trianglelefteq_1, \overline{x}) > K(\trianglelefteq_2, \overline{x})$ for every infinite sequence $\overline{x}$.

This is a rather strong definition of lenience: in practice, we are also interested in well-quasi-orders that *tend* to be more lenient than others on commonly-encountered sequences. However, this definition will suffice for this paper.

4. Termination combinators

In this section we describe the primitive combinators provided by our library, and prove that they construct correct WQOs.

4.1 The trivial test

The simplest WQO is one that relates everything, and hence blows the whistle immediately:

```
alwaysT :: TTest a
alwaysT = WQO (λx y → True)
```

This $alwaysT$ is trivially correct, and not at all lenient. Nonetheless, it can be usefully deployed as a "placeholder" well-quasi-order when we have yet to elaborate a well-quasi-order, or a natural well-quasi-order does not exist (e.g. consider well-quasi-ordering values of type $IO\ Int$).

4.2 Termination for finite sets

Our next combinator deals with termination over finite sets:

```
finiteT :: ∀ a.Finite a ⇒ TTest a
finiteT = WQO (≡)

class Eq a ⇒ Finite a where
  elements :: [ a ]   -- Members of the type
```

This WQO relates equal elements, leaving unequal elements unrelated. Provided all the elements are drawn from a finite set, $(\equiv)$ is indeed a WQO:

Proof. Consider an arbitrary sequence $\overline{x}^\infty \in A^\infty$ where there are a finite number of elements of A. Since A is finite, the sequence must repeat itself at some point — i.e. $\exists jk.j \neq k \land x_j = x_k$. The existence of this pair proves that $finiteT$ defines a well-quasi-order. Meanwhile, transitivity follows trivially from the transitivity of $(\equiv)$. □

Using $finiteT$, we can trivially define the $boolT$ combinator used in the introduction:

```
boolT :: TTest Bool
boolT = finiteT
```

The combinator *finiteT* is polymorphic. The fact that the element type a must be finite using the "$Finite\ a\ \Rightarrow$" constraint in *finiteT*'s type. But there is clearly something odd here. First, 'finiteT' does not use any methods of class $Finite$, and second, it is the the client who makes a new type T into an instance of $Finite$, and the library has no way to check that the instance is telling the truth. For example, a client could bogusly say:

```
instance Finite Integer where
  elements = [ ]
```

Moreover, the user could give a bogus implementation of equality:

```
data T = A | B
instance Eq T where
  (≡) p q = False
istance Finite T where
  elements = [A, B]
```

Here the new type T is finite, but since the equality function always returns *False*, the whistle will never blow.

So our library guarantees the soundness of the termination testers *under the assumption that the instances of certain classes at the element type A satisfy corresponding correctness conditions*. Specifically:

- $(\equiv)$ must be reflexive and transitive at type A.
- The type A must have only a finite number of distinct elements (distinct according to $(\equiv)$, that is).

Another way to say this is that the instances of Eq and $Finite$ form part of the trusted code base. This is not unreasonable. On the one hand, these proof obligations are simple for the programmer to undertake — much, much simpler than proving that a particular boolean-valued function is a WQO.

On the other hand, it is unrealistic for the library to check that $elements$ is a finite list and that the two values we compare are elements of that finite list, for instance, by using runtime assertions. In the example of Section 3.2 there are 2^{26} elements of the type $Set\ Char$, so making these checks at runtime would be a very bad idea.

4.3 Termination for well-ordered sets

Another very useful primitive well-quasi-order is that on elements drawn from well-ordered sets: every well-order is a well-quasi-order (but clearly not vice-versa):

```
wellOrderedT :: WellOrdered a ⇒ TTest a
wellOrderedT = WQO (≤)
class Ord a ⇒ WellOrdered a
```

Similar to $Finite$, the $WellOrdered$ predicate picks out types with least elements; that is ones have a total order (hence the Ord superclass) and a least element. The client's proof obligations about instances of a type A are:

- $(\leq)$ defines a total order (i.e. it is antisymmetric, transitive and total)
- For every (possibly infinite) non-empty set $X \subseteq A$ of elements, $\exists (y :: A) \in X. \forall (x :: A) \in X. y \leq x$.

Under these conditions, $(\leq)$ is a WQO:

Proof. Transitivity is immediate by assumption. Now consider an arbitrary sequence $\overline{x}^{\infty}$. Each pair of adjacent elements x_j, x_{j+1} in the sequence is either shrinking (so $\neg(x_j \leq x_{j+1})$) or non-decreasing (so $x_j \leq x_{j+1}$). If we have at least one pair of the latter kind, the well-quasi-order property holds. The dangerous possibility is that all our pairs may be of the former sort.

Because we have that $\forall j. \neg(x_j \leq x_{j+1})$, by the reflexivity of $\leq$ we know that $\forall j. \neg(x_j < x_{j+1})$ — i.e. we have an infinitely descending chain. However, this fact contradicts the assumption that $\leq$ is a well-order. ☐

Given $wellOrderedT$ and an instance $WellOrdered\ Int$, it is trivial to define a suitable $intT$ (as used in the introduction):

```
intT :: TTest Int
intT = wellOrderedT
```

4.4 Functorality of termination tests

Now that we have defined a number of primitive termination tests, we are interested in defining some combinators that let us combine these tests into more powerful ones. The first of these shows that $TTest$ is a contravariant functor:

```
class Cofunctor f where
  cofmap :: (b → a) → f a → f b
instance Cofunctor TTest where
  cofmap f (WQO (⊴)) = WQO $ λx y → f x ⊴ f y
```

So, for example, here is how a client could build a (not very good) termination test for labelled rose trees:

```
data Tree = Node Label [Tree]
size :: Tree → Int
size (Tree n ts) = 1 + sum (map size ts)
treeT :: TTest Tree
treeT = cofmap size wellOrderedT
```

Here we use $size$ to take the size of a tree, and use the fact that Int is well-ordered by $\leq$ as the underlying termination test.

The defining laws of contravariant functors (cofunctors) are:

1. Identity: $cofmap\ id = id$
2. Composition: $cofmap\ f \circ cofmap\ g = cofmap\ (g \circ f)$

These two laws are easy to verify for $TTest$ instance above. Similarly, it is easy to to show that $(cofmap\ f\ t)$ is a well-quasi-order if t is.

Intuitively, the reason that $TTest$ is a *contravariant* functor is that it $TTest\ a$ is a *consumer* rather than a producer of values of type a. For the same reason, the arrow type $(\rightarrow)$ is contravariant in its first type argument.

In section Section 6.2, we show how this definition of $cofmap\ f$ can be improved.

4.5 Termination for sums

We are able to build termination test for sum types, given tests for the components:

```
eitherT :: TTest a → TTest b → TTest (Either a b)
eitherT (WQO (⊴a)) (WQO (⊴b)) = WQO (⊴)
  where
    (Left a1)  ⊴ (Left a2)  = a1 ⊴a a2
    (Right b1) ⊴ (Right b2) = b1 ⊴b b2
    _          ⊴ _          = False
```

The ordering used here treats elements from the same side of the sum (i.e. both *Left* or both *Right*) using the corresponding component ordering, and otherwise treats them as unordered.

Does this test define a WQO? Yes:

Proof. Consider an arbitrary sequence $\overline{x}^{\infty} \in (Either\ A\ B)^{\infty}$. Form the subsequences $\overline{a}^{\infty} = \{a_i \in A \mid Left\ a_i \in \overline{x}^{\infty}\}$ and $\overline{b}^{\infty} = \{b_i \in B \mid Right\ b_i \in \overline{x}^{\infty}\}$. Since the x sequence is infinite, at least one of these subsequences must be infinite. Without loss of generality, assume that the $\overline{a}^\infty$ sequence is infinite. Now, the fact that $eitherT\ wqoa\ wqob$ is a well-quasi-order follows directly from the fact that $wqoa$ is a well-quasi-order. □

Incidentally, notice that if the component types are both $()$, the test boils down to the same as the finite-set test for $Bool$ in Section 4.2. Conversely, it is straightforward (albeit inefficient) to define $finiteT$ by iterating $eitherT$ once for each item in the $elements$ list, and the reader is urged to do so as an exercise.

The test $eitherT\ \ t_1\ \ t_2$ is at least as lenient as t_1 or t_2 (in the sense of Definition 3.4), and is often strictly more lenient. Specifically, if $\overline{x} \in Either\ A\ B$, and $L(\overline{x}) = \{x \mid Left\ x \in \overline{x}\}$, and similarly for $R(\overline{x})$, then

$$\begin{array}{l} min(K(t_1, L(\overline{x})),\ K(t_2, R(\overline{x}))) \\ \quad \leq \quad K(Either\ t_1\ t_2, \overline{x}) \\ \leq \quad K(t_1, L(\overline{x})) + K(t_2, R(\overline{x})) \end{array}$$

Both the upper and lower bounds of this inequality can actually be realised. For example, with the test

$$eitherT\ finiteT\ finiteT :: TTest\ (Either\ ()\ Bool)$$

the lower bound is realised by $\overline{x}^\infty = L\ (), L\ (), L\ (), \ldots$, and the upper bound by $\overline{x}^\infty = L\ (), R\ True, R\ False, L\ (), R\ True, \ldots$.

Although we haven't defined many combinators, we already have enough to be able to define natural well-quasi-orders on many simple data types. For example, we can well-quasi-order $Maybe\ T$ if we can well-quasi-order T itself:

```
maybeT :: TTest a → TTest (Maybe a)
maybeT wqo = cofmap inject (eitherT alwaysT wqo)
  where
    inject Nothing = Left ()
    inject (Just x) = Right x
```

To define $maybeT$ we have adopted a strategy — repeated later in this document — of "injecting" the $Maybe$ data type (which our combinators cannot yet handle) into a simpler data type which is handled by a primitive combinator — in this case, $Either$[2].

Note that we use $alwaysT$ from Section 4.1 to well-quasi-order values of unit type — there really is no non-trivial way to order a type with only one value.

4.6 Termination for products

Just like we could for sum types, we can define a combinator for well-quasi-ordering product types, given WQOs on the component types:

```
pairT :: TTest a → TTest b → TTest (a, b)
pairT (WQO (⊴_a)) (WQO (⊴_b)) = WQO (⊴)
  where
    (a₁, b₁) ⊴ (a₂, b₂) = (a₁ ⊴_a a₂) ∧ (b₁ ⊴_b b₂)
```

The fact that $pairT$ defines a WQO is quite surprising. We can assume that $\trianglelefteq_a$ and $\trianglelefteq_b$ are WQOs, but that only means that given input sequences $\overline{a}^\infty$ and $\overline{b}^\infty$ respectively, there exists some $i < j.\, a_i \trianglelefteq_a a_j$ and $k < l.\, b_k \trianglelefteq_b b_l$. Yet for $pairT$ to define a WQO there must exist a $p < q$ such that $a_p \trianglelefteq_a a_q$ and *simultaneously* $b_p \trianglelefteq_b b_q$. How can we know that the related elements of the two sequences will ever "line up"?

Nonetheless, it is indeed the case, as the following proof demonstrates. First we need a lemma:

Lemma 4.1. *For any well-quasi-order* $\trianglelefteq \in A \times A$ *and* $\overline{x}^\infty \in A^\infty$, *there exists some* $n \geq 0$ *such that* $\forall j > n.\, \exists k > j.\, x_j \trianglelefteq x_k$.

[2] In this and many other examples, the Glasgow Haskell Compiler's optimisation passes ensure that the intermediate $Either$ value is not actually constructed at runtime.

This lemma states that, beyond some some threshold value n, *every* element x_j (where $j > n$) has a related element x_k somewhere later in the sequence.

Proof. This lemma can be shown by a Ramsey argument. Consider an arbitrary sequence $\overline{x}^\infty$. Consider the sequence

$$\overline{y} = \{x_i \mid x_i \in \overline{x}^\infty, \forall j > i.\, \neg(x_i \trianglelefteq x_j)\}$$

of elements of $\overline{x}^\infty$ which are embedded into no later element. If this sequence was infinite it would violate the well-quasi-order property, since by definition none of the elements of the sequence are related by $\trianglelefteq$. Hence we have a constructive proof of the proposition if we take n to be $\max\{i \mid x_i \in \overline{y}\}$. □

A proof of the fact that $pairT$ defines a well-quasi-order as long as its two arguments does — a result that e.g. Kruskal [1] calls the Cartesian Product Lemma — now follows:

Proof. Consider an arbitrary sequence $\overline{(a,b)}^\infty \in (A \times B)^\infty$. By Lemma 4.1, there must be a n such that $\forall j > n. \exists k > j. a_j \trianglelefteq_a a_k$. Hence there must be at least one infinite subsequence of $\overline{a}^\infty$ where adjacent elements are related by $\trianglelefteq_a$ — i.e. $a_n \trianglelefteq_a a_{l_0} \trianglelefteq_a a_{l_1} \trianglelefteq_a \ldots$ where $n < l_0 < l_1 < \ldots$.

Now form the infinite sequence $b_j, b_{l_0}, b_{l_1} \ldots$. By the properties of $\trianglelefteq_b$, there must exist some m and n such that $m < n$ and $b_{l_m} \trianglelefteq_b b_{l_n}$. Because $\trianglelefteq_a$ is transitive, we also know that $a_{l_m} \trianglelefteq_a a_{l_n}$.

This inference, combined with the fact that $\trianglelefteq_a$ and $\trianglelefteq_b$ are valid WQOs, and that transitivity follows by the transitivity of both the component WQOs, proves that $pairT\ \trianglelefteq_a \trianglelefteq_b$ is a well-quasi-order. □

From a leniency point of view, we have a lower bound on the leniency of a test built with $pairT$:

$$\max\left(\begin{array}{l} K(t_1, \{a_i \mid (a_i, b_i) \in \overline{x}^\infty\}), \\ K(t_2, \{b_i \mid (a_i, b_i) \in \overline{x}^\infty\}) \end{array} \right) \leq K(pairT\ t_1\ t_2, \overline{x}^\infty)$$

However, there is no obvious upper bound on the characteristic index. Not even

$$K(t_1, \{a_i \mid (a_i, b_i) \in \overline{x}^\infty\}) * K(t_2, \{b_i \mid (a_i, b_i) \in \overline{x}^\infty\})$$

is an upper bound for the characteristic index of $pairT\ wqoa\ wqob$ — for example, the proposed upper bound is violated by the well-quasi-order $pairT\ finiteT\ wellOrderedT$ and the sequence $(T, 100), (F, 100), (T, 99), (F, 99), ..., (F, 0)$, which has characteristic index 300, despite the component characteristic indexes being 2 and 1 respectively.

We now have enough combinators to build the string termination test from Section 3.2:

```
stringT :: TTest String
stringT = cofmap inject (pairT finiteT wellOrderedT)
  where inject s = (mkSet s, length s)
```

We assume a type of sets with the following interface:

```
instance (Ord a, Finite a) ⇒ Finite (Set a) where ...
mkSet :: Ord a ⇒ [a] → Set a
```

(We use the bounded Int length of a string in our $stringT$, but note that this would work equally well with a hypothetical type of unbounded natural numbers Nat, should you define a suitable $WellOrdered\ Nat$ instance.)

The big advantage in defining $stringT$ with our combinator library is that Theorem 3.2 in Section 3.2 is not needed: the termination test is sound by construction, provided only that (a) there are only a finite number of distinct sets of characters, and (b) the Ints are well ordered.

4.7 Finite maps

It is often convenient to have termination tests over finite mappings, where the domain is a finite type — for example, we will need such a test in Section 7.4. One way to implement such a test is to think of the mapping as a large (but bounded) arity tuple. To compare $m1$ and $m2$, where $m1$ and $m2$ are finite maps, you may imagine forming two big tuples

```
(lookup k1 m1, lookup k2 m1, ..., lookup kn m1)
(lookup k1 m2, lookup k2 m2, ..., lookup kn m2)
```

where $k1...kn$ are all the elements of the key type. The *lookup* returns a *Maybe* and, using the rules for products (Section 4.6), we return *False* if any of the constructors differ; that is, if the two maps have different domains. If the domains are the same, we will simply compare the corresponding elements pairwise, and we are done.

We can implement this idea as a new combinator, *finiteMapT*. We assume the following standard interface for finite maps:

```
assocs :: Ord k ⇒ Map k v → [(k, v)]
keysSet :: Ord k ⇒ Map k v → Set k
elems  :: Ord k ⇒ Map k v → [v]
lookup :: Ord k ⇒ k → Map k v → Maybe v
```

From which the combinator follows:

```
finiteMapT :: ∀k v.(Ord k, Finite k)
              ⇒ TTest v → TTest (Map k v)
finiteMapT (WQO (⊴)) = WQO test
  where
    test :: Map k v → Map k v → Bool
    test m1 m2 = keysSet m1 ≡ keysSet m2
                 ∧ all (ok m1) (assocs m2)
    ok :: Map k v → (k, v) → Bool
    ok m1 (k2, v2) = case lookup k2 m1 of
      Just v1 → v1 ⊴ v2
      Nothing → error "finiteMapT"
```

In fact, the *finiteMapT* combinator can be defined in terms of our existing combinators, by iterating the *pairT* combinator (we also make use of *maybeT* from Section 4.5):

```
finiteMapT_indirect :: ∀k v.(Ord k, Finite k)
                       ⇒ TTest v → TTest (Map k v)
finiteMapT_indirect wqo_val
   = go (const ()) finiteT elements
  where
    go :: ∀vtup.(Map k v → vtup) → TTest vtup → [k]
       → TTest (Map k v)
    go acc test [] = cofmap acc test
    go acc test (key : keys)
       = go acc′ (pairT (maybeT wqo_val) test) keys
      where acc′ mp = (lookup key mp, acc mp)
```

Unfortunately, this definition involves enumerating all the elements of the type (via the call to *elements*), and there might be an unreasonably large number of such elements, even though any particular *Map* might be small. For these reasons we prefer the direct implementation.

5. Termination tests for recursive data types

Now that we have defined well-quasi-order combinators for both sum and product types, you may very well be tempted to define a WQO for a data type such as lists like this:

```
list_bad :: ∀a.TTest a → TTest [a]
list_bad test_x = test_xs
  where
    test_xs :: TTest [a]
    test_xs = cofmap inject (eitherT finiteT
                                  (pairT test_x test_xs))
    inject []       = Left ()
    inject (y : ys) = Right (y, ys)
```

Unfortunately the *list_bad* combinator would be totally bogus. Notice that *list_bad* only relates two lists if they have exactly the same "spines" (i.e. their lengths are the same) — but unfortunately, there are infinitely many possible list spines. Thus in particular, it would be the case that the following infinite sequence would be accepted by the (non!) well-quasi-order *list_bad finite*:

$$[\,], [()], [(), ()], [(), (), ()], \ldots$$

We would like to prevent such bogus definitions, to preserve the safety property of our combinator library. The fundamental problem is that *list_bad* isn't well-founded in some sense: our proof of the correctness of *cofmap*, *eitherT* and so on are sufficient to show only that *test_xs* is a well-quasi-order if and only if *test_xs* is a well-quasi-order — a rather uninformative statement! This issue fundamentally arises because our mathematics is set-theoretical, whereas Haskell is a language with complete partial order (cpo) semantics.

Our approach is to rule out such definitions by making all of our combinators *strict in their well-quasi-order arguments*. Note that we originally defined *TTest* using the Haskell **data** keyword, rather than **newtype**, which means that all the combinator definitions presented so far are in fact strict in this sense. This trick means that the attempt at recursion in *list_bad* just builds a loop instead — $\forall w.list\_bad\ w = \bot$.

It is clear that making our well-quasi-order combinators non-strict — and thus allowing value recursion — immediately makes the combinator library unsafe. However, we still need to be able to define well-quasi-orders on recursive data types like lists and trees, which — with the combinators introduced so far — is impossible without value-recursion. To deal with recursive data types, we need to introduce an explicit combinator for reasoning about fixed points in a safe way that is *lazy* in its well-quasi-order argument, and hence can be used to break loops that would otherwise lead to divergence.

5.1 Well-quasi-ordering any data type

You might wonder if it is possible to naturally well-quasi-order recursive data types at all. To show that we can, we consider well-quasi-ordering a "universal data type", *UnivDT*:

```
data UnivDT = U String [UnivDT]
```

The idea is that the *String* models a constructor name, and the list the fields of the constructor. By analogy with real data types, we impose the restrictions that there are only a finite number of constructor names, and for any given constructor the length of the associated list is fixed. In particular, the finite list of constructors will contain "Nil" (of arity 0) and "Cons" (of arity 2), with which we can model the lists of the previous section.

We can impose a well-quasi-order on the suitably-restricted data type *UnivDT* like so:

```
univT :: TTest UnivDT
univT = WQO test
  where test u1@(U c1 us1) (U c2 us2)
          = (c1 ≡ c2 ∧ and (zipWith test us1 us2)) ∨
            any (u1‘test‘) us2
```

Elements $u1$ and $u2$ of *UnivDT* are related by the well-quasi-order if either:

- The constructors $c1$ and $c2$ match, and all the children $us1$ and $us2$ match (remember that the length of the list of children is fixed for a particular constructor, so $us1$ and $us2$ have the same length). When this happens, the standard terminology is that $u1$ and $u2$ *couple*.
- The constructors don't match, but $u1$ is related by the well-quasi-order to one of the children of $u2$. The terminology is that $u1$ *dives* into $u2$.

Although not immediately obvious, this test does indeed define a well-quasi-order on these tree-like structures (the proof is similar to that we present later in Section 5.2), and it is this well-quasi-order (sometimes called the "homeomorphic embedding") which is used in most classical supercompilation work (see e.g. [4]).

Once again, we stress that for this test to be correct, the constructor name must determine the number of children: without this assumption, given at least two constructors F and G you can construct a chain such as

```
U "F" [ ], U "F" [U "G" [ ]], U "F" [U "G" [ ], U "G" [ ]], ...
```

which is not well-quasi-ordered by the definition above.

5.2 Well-quasi-ordering functor fixed points

We could add the well-quasi-order on our "universal data type" as a primitive to our library. This would be sufficient to allow the user to well-quasi-order their own data types – for example, we could define an ordering on lists as follows:

```
list_univ :: TTest [UnivDT]
list_univ = cofmap to_univ univT
to_univ :: [UnivDT] → UnivDT
to_univ [ ]      = U "Nil" [ ]
to_univ (x : xs) = U "Cons" [x, to_univ xs]
```

However, this solution leaves something to be desired: for one, we would like to be able to well-quasi-order lists $[a]$ for an arbitrary element type a, given a well-quasi-ordering on those elements. Furthermore, with this approach there is scope for the user to make an error in writing to_univ which violates the invariants on the $UnivDT$ type. This would break the safety promises of the well-quasi-order library.

We propose a different solution that does not suffer from these problems. The first step is to represent data types as fixed points of functors in the standard way. For example, lists are encoded as follows:

```
newtype Fix t = Roll {unroll :: t (Fix t)}
data ListF a rec = NilF | ConsF a rec
  deriving (Functor, Foldable, Traversable)
fromList :: [a] → Fix (ListF a)
fromList [ ]      = Roll NilF
fromList (y : ys) = Roll (ConsF y (fromList ys))
```

The $fixT$ ***combinator*** Our library then provides a single primitive that can be used to well-quasi-order any data type built out of this sort of explicit fixed point scheme:

```
fixT :: ∀t.Functor t
      ⇒ (∀rec.t rec → [rec])
      → (∀rec.t rec → t rec)
      → (∀rec.TTest rec → TTest (t rec))
      → TTest (Fix t)
fixT kids p f = wqo
  where
    wqo = WQO (λ(Roll a) (Roll b) → test a b)
    test a b = (⊴) (f wqo) (p a) (p b) ∨
               any (test a ∘ unroll) (kids b)
```

The arguments of $fixT$ are as follows:

- A type constructor $t :: * \to *$ that is equipped with the usual functorial lifting function $fmap :: \forall a\ b.(a \to b) \to t\ a \to t\ b$. (By chance, we do not in fact use $fmap$ in our definition, though it will show up in our proof that $fixT$ is correct. Alternative representations for $TTest$ — such as that discussed in Section 6.2 — may indeed use $fmap$ in their definition of $fixT$.)
- A function $kids$ with which to extract the (or some of the) "children" of a functor application.
- A function p that we will call the *calibrator* whose purpose is to map elements of type $t\ rec$ to elements of type $t\ rec$ but where the holes in the returned shape are filled in with elements returned from the $kids$ function. We explain this in detail later in this section.
- Finally, a function which determines how we will create a well-quasi-order $t\ rec$ given a well-quasi-order for some arbitrary rec. The only invariant we require on this is that if given a correct well-quasi-order it returns a correct well-quasi-order. This invariant will be trivially satisfied as long as the user constructs all $TTest$s using the combinators of our library.

The definition of `test` in $fixT$ is analogous to the test we saw in $univT$ — the first argument of $\vee$ tests whether the left side couples with the right, and the second argument determines whether the left side dives into one of the $kids$ of the right. The coupling case is actually slightly more general than the coupling we have seen until now, due to the calibrator p being applied to a and b before we compare them.

We now present the preconditions for $fixT$ to define a well-quasi-order.

Definition 5.1 ($fixT$ preconditions)**.** For a particular type constructor $t :: * \to *$ equipped with the usual $fmap :: \forall a\ b.(a \to b) \to t\ a \to t\ b$, and functions $kids$, p and f (suitably typed) we say that they jointly satisfy the $fixT$ preconditions if:

- All elements x of type $Fix\ t$ must be finite, in the sense that $size\ x$ is defined, where $size$ is as follows:

```
size :: Fix t → Integer
size = (1+) ∘ sum ∘ map size ∘ kids ∘ unroll
```

- The calibrator function p must satisfy the (non-Haskell) dependent type:

$$g :: (y : t\ a) \to t\ \{x : a \mid x \in kids\ y\}$$

The first condition is not interesting[3] – it ensures that we can't be calling $kids$ forever while comparing two elements. The second condition is the interesting one. Typically one thinks of $kids$ as returning *all* the children of a functor. For instance, consider the $BTreeF$ functor below, that defines labelled binary trees:

```
data BTreeF a rec = BNil | BNode a rec rec
kids_tree :: ∀a rec.BTreeF a rec → [rec]
kids_tree BNil            = [ ]
kids_tree (BNode _ x y)   = [x, y]
```

In this case, a valid calibrator is simply the identity

```
p :: ∀a rec.BTreeF a rec → BTreeF a rec
p BNil          = BNil
p (BNode a x y) = BNode a x y
```

[3] Again, this constraint arises from our attempt to use Haskell (a language with cpo semantics) as if it had set semantics.

since both x and y are returned by $kids\_tree$. Consider however, a different version of $kids$ that only returns the left branch of a node:

```
kids_tree_alt :: ∀ a rec.BTreeF a rec → [rec]
kids_tree_alt BNil          = [ ]
kids_tree_alt (BNode _ x y) = [x]
```

A valid calibrator for this $kids\_tree\_alt$ can only plug in the holes of the functor elements that can be returned from $kids\_tree\_alt$. Consider:

```
p_ok, p_bad :: BTreeF a rec → BTreeF a rec
p_ok BNil          = BNil
p_ok (BNode a x y) = BNode a x x

p_bad BNil          = BNil
p_bad (BNode a x y) = BNode a x y
```

In this example p_ok is a valid calibrator, as it only uses x, which belongs in $kids\_tree\_alt\ (BNode\ a\ x\ y)$. However p_bad is not a valid calibrator as it uses y, which is not returned by $kids\_tree\_alt$. So, the role of the calibrator is to correct the behaviour of the test, depending on the implementation of $kids$.

Arguably, the extra generality of a $kids$ function that does not return all kids or may have even more exotic behaviour is rarely used but provides for an elegant generic proof of correctness of $fixT$.

Using* $fixT$ *with lists One correct way to use the $fixT$ combinator is with the following $kids\_list$ function

```
kids_list :: ∀ rec.ListF a rec → [rec]
kids_list NilF        = [ ]
kids_list (ConsF _ xs) = [xs]
```

along with the identity calibrator to define a correct-by-construction well-quasi-order for lists (realising the "Finite Sequence Theorem" of Kruskal [1]):

```
listT :: ∀ a.TTest a → TTest [a]
listT wqo_elt
  = cofmap fromList (fixT kids_list id wqo_fix)
  where
    wqo_fix :: ∀ rec.TTest rec → TTest (ListF a rec)
    wqo_fix wqo_tail
      = cofmap inject $
          eitherT finiteT (wqo_elt `pairT` wqo_tail)

    inject :: ∀ rec.ListF a rec → Either () (a, rec)
    inject NilF          = Left ()
    inject (ConsF y ys) = Right (y, ys)
```

Is* $fixT$ *correct? Now we have seen an example of the use of $fixT$, we are in a position to tackle the important question as to whether it actually defines a well-quasi-order:

Theorem 5.1 (Correctness of $fixT$)**.** *If the preconditions of fixT (Definition 5.1) are satisfied then* $fixT\ \ kids\ \ p\ \ f$ *defines a well-quasi-order.*

Proof. By contradiction, assume that under our assumptions, there exists at least one accepted infinite sequence $\in (Fix\ t)^\infty$ for the relation $(\trianglelefteq)\ (fixT\ kids\ p\ f)$.

We pick the minimal such accepted sequence $\overline{t}^\infty$, such that for all $n \in \mathbb{N}$ and accepted sequences $\overline{s}^\infty$ such that $\forall i.0 \le i < n.t_i = s_i$, we have that $size\ t_n \le size\ s_n$.

We now form the possibly infinite set of children, D:

$$D = \{k \mid i \in \mathbb{N}, k \in kids\ (unroll\ t_i)\}$$

As a subgoal, we claim that $fixT\ \ kids\ \ p\ \ f :: TTest\ \ D$ is a WQO. In other words, the union of all children of the minimal sequence is well-quasi ordered by $fixT\ \ kids\ \ p\ \ f$. To see this, we proceed by contradiction: assume there is some accepted infinite sequence $\overline{r}^\infty \in D^\infty$. Because each $kids\ \ (unroll\ t_i)$ is finite (since $size\ \ t_i$ is finite), the accepted sequence $\overline{r}^\infty$ must have an infinite subsequence $\overline{q}^\infty$ such that $q_i \in kids\ \ (unroll\ \ t_{f(i)})$ for some f such that $\forall j.f(0) \le f(j)$. Given such a $\overline{q}^\infty$, we can define a new infinite sequence $\overline{s}^\infty \in (Fix\ t)^\infty$:

$$\overline{s}^\infty = t_0, t_1, \ldots, t_{f(0)-1}, q_{f(0)}, q_{f(1)}, \ldots$$

The sequence $\overline{s}^\infty$ must be accepted because otherwise, by the definition of $fixT$ the original $\overline{t}^\infty$ would be rejected (by the "dive" rule). But if it is accepted then we have a contradiction to the minimality of $\overline{t}^\infty$ since $size\ \ q_{f(0)}\ \ <\ \ size\ \ t_{f(0)}$, $q_{f(0)} \in kids\ \ (unroll\ t_{f(0)})$, and the children of an element have smaller size than their parent. We conclude that $fixT\ \ kids\ \ p\ f$ is a WQO.

This fact means that $f\ \ (fixT\ \ kids\ \ p\ \ f) :: TTest\ \ (t\ \ D)$ is a WQO. Consider now the infinite minimal sequence $\overline{t}^\infty$ again and the mapping of each element through the calibrator p: $u_i = p\ (unroll\ t_i)$. Each u_i has type: $u_i :: t\ \{x \mid x \in kids\ t_i\}$. Furthermore, because t is a functor and $\forall i.kids\ t_i \subseteq D$, we have that $u_i\ ::\ t\ \{x\ \mid\ x\ \in\ D\}$ and hence we have an infinite sequence of elements of type $t\ \ D$. Hence there exist two elements $p\ \ (unroll\ t_i)$ and $p\ \ (unroll\ t_j)$ such that they are related in the WQO $f\ \ (fixT\ \ kids\ \ p\ \ f)$. By the definition of $fixT$, this contradicts the initial assumption that the sequence $\overline{t}^\infty$ is accepted by $fixT\ kids\ p\ f$. □

Our proof is essentially a proof of the Tree Theorem [1] to our setting, though the proof itself follows the simpler scheme in Nash-Williams [5].

Generality is good, but the calibrator has an complex type which may be somewhat hard for Haskell programmers to check. In the next section we show how $kids$ and the calibrator p can be written generically, and hence can be entirely eliminated from the preconditions for $fixT$.

Further remarks on lists Inlining our combinators and simplifying, we find that our earlier definition of $listT$ is equivalent to the following:

```
listT' :: TTest a → TTest [a]
listT' (WQO (⊴)) = WQO go
  where
    go (x : xs) (y : ys)
       | x ⊴ y, go xs ys = True
       | otherwise       = go (x : xs) ys
    go (_: _)   [ ]      = False
    go [ ]      [ ]      = True
    go [ ]      (_ : ys) = go [ ] ys
```

It is interesting to note that $listT'$ could be more efficient:

- By noticing that $\forall ys.go\ [\,]\ ys = True$, the last clause of go can be replaced with $go\ [\,]\ (\_{:}ys) = True$. This avoids a redundant deconstruction of the list in the second argument (at the cost of changing the meaning if the second argument is in fact infinite — a possibility we explicitly excluded when defining $fixT$).
- By noticing that $\forall x, xs, ys.go\ (x : xs)\ ys \implies go\ xs\ ys$, the first clause of go can avoid falling through to test $go\ (x{:}xs)\ ys$ if it finds that $go\ xs\ ys \equiv False$.

Both of these observations are specific to the special case of lists: for other data types (such as binary trees) $fixT$ will generate an implementation that does not have any opportunity to apply these "obvious" improvements.

5.3 From functors to *Traversable*s

As we have presented it, the user of $fixT$ still has the responsibility of providing a correct $kids$ and a calibrator p with a strange dependent type (which Haskell does not even support!). Happily, we can greatly simplify things by combining the recently-added ability of the Glasgow Haskell Compiler [6] to automatically derive $Traversable$ instances. The $Traversable$ [7] type class allows us to write the following:

```
kids_traverse :: ∀t a. Traversable t ⇒ t a → [a]
kids_traverse = unGather ∘ traverse (λx → Gather [x])
newtype Gather a b = Gather {unGather :: [a]}
instance Functor (Gather a) where
  fmap _ (Gather xs) = Gather xs
instance Applicative (Gather a) where
  pure x = Gather [ ]
  Gather xs ⟨*⟩ Gather ys = Gather (xs ++ys)
```

It follows from the $Traversable$ laws that $kids_{traverse}$ collects "all the children" of $t\ rec$, and as a consequence (See Section 4.1 of [7]) the corresponding projector is just id. We can therefore satisfy the preconditions of Definition 5.1 by setting:

$$\begin{array}{lcl} kids & := & kids_{traverse} \\ p & := & id \end{array}$$

The corresponding generic definition $gfixT$ becomes:

```
gfixT :: Traversable t
      ⇒ (∀rec. TTest rec → TTest (t rec))
      → TTest (Fix t)
gfixT = fixT kids_traverse id
```

Therefore, if the user of the library has a correct $Traversable$ instance (possibly compiler-generated), they need not worry about the calibrator or $kids$ functions at all, and cannot violate the safety guarantees of the library.

6. Optimisation opportunities

Having defined our combinators, we pause here to consider two optimisations we can apply to our definitions. Thanks to our clearly-defined abstract interface to the $TTest$ and $History$ types these optimisations are entirely transparent to the user.

6.1 Pruning histories using transitivity

In this section we consider an improvement to the definition of $initHistory$ in Section 3.1.

Normally, whenever a $History\ a$ receives a new element $x' :: a$ to compare against its existing $\overline{x}^n$, we test all elements to see if $\exists i < n.x_i \trianglelefteq x'$. If we do not find such an i, we append x' to form the new sequence $\overline{x'}^{n+1}$ which will be tested against subsequently. Thus at every step the number of tests that need to be done grows by one.

There is an interesting possibility for optimisation here: we may in fact exclude from $\overline{x'}$ any element x_j $(0 \le j < n)$ such that $x' \trianglelefteq x_j$. The reason is that if a later element $x'' :: a$ is tested against $\overline{x'}$, then by transitivity of $\trianglelefteq$, $x_j \trianglelefteq x'' \implies x' \trianglelefteq x''$ — thus it is sufficient to test x'' only against x', skipping the test against the "older" element x_j entirely.

To actually make use of this optimisation in our implementation, our implementation must (for all $0 \le j < n$), test $x' \trianglelefteq x_j$ as well as $x_j \trianglelefteq x'$. To make this test more efficient, we could redefine $TTest$ so when evaluated on x and y it returns a pair of $Bool$ representing $x \trianglelefteq y$ and $y \trianglelefteq x$ respectively:

```
data TTest a = WQO {(⊴) :: a → a → (Bool, Bool)}
```

Returning a pair of results improves efficiency because there is almost always significant work to be shared across the two "directions".

A version of the core data types improved by this new $TTest$ representation and the transitivity optimisation is sketched below:

```
data TTest a = WQO (a → a → (Bool, Bool))
newtype History a = H {test :: a → TestResult a}
initHistory :: ∀a. TTest a → History a
initHistory (WQO (⊴)) = H (go [ ])
  where
    go :: [a] → a → TestResult a
    go xs x
      | or gts = Stop
      | otherwise
      = Continue (H (go (x : [x | (False, x) ← lts ‘zip‘ xs])))
        where (gts, lts) = unzip (map (⊴ x) xs)
```

It is unproblematic to redefine all of our later combinators for the elaborated $TTest$ type so we can take advantage of this transitivity optimisation.

6.2 Making *cofmap* more efficient

The alert reader may wonder about how efficient the definition of $cofmap$ in Section 4.4 is. Every use of a WQO of the form $cofmap\ f\ wqo$ will run f afresh on each of the two arguments to the WQO. This behaviour might lead to a lot of redundant work in the implementation of $test$ (Section 3.1), as repeated uses of $test$ will repeatedly invoke the WQO with the same first argument. By a change of representation inside the library, we can help ensure that this per-argument work is cached and hence only performed once for each value presented to $test$:

```
data TTest a where
  TT :: (a → b) → (b → b → Bool) → TTest a
newtype History a = H {test :: a → TestResult a}
initHistory :: TTest a → History a
initHistory (TT f (⊴)) = H (go [ ])
  where
    go fxs x
      | any (⊴ fx) fxs = Stop
      | otherwise      = Continue (H (go (fx : fxs)))
     where fx = f x
```

A $History$ now includes a function f mapping the client's data a to the maintained history list $[b]$. When testing, we apply the function to get a value $fx :: b$, which we compare with the values seen so far.

With this new representation of $TTest$, $cofmap$ may be defined as follows:

```
instance Cofunctor TTest where
  cofmap f (WQO prep (⊴)) = WQO (prep ∘ f) (⊴)
```

The ability to redefine $TTest$ to be more than simply a WQO is one of the reasons why we distinguish "termination tests", which the client builds using the combinators, and "WQOs" which are part of the implementation of a termination test, and are hidden from the client.

All the $TTest$-using code we present is easily adapted for the above, more efficient, representation of $TTest$. Furthermore, this technique can further be combined with the optimisation described in Section 6.1 with no difficulties.

7. Supercompilation termination tests

Now that we have defined a combinator library for termination tests, you might wonder whether it is actually general enough to capture those tests of interest in supercompilation. In this section, we demonstrate that this is so.

7.1 Terminating evaluators

Before we discuss those well-quasi-orders used for supercompilation, we would like to motivate them with an example of their use.

A supercompiler is, at its heart, an evaluator, and as such it implements the operational semantics for the language being supercompiled. However, the language in question is usually Turing complete, and we would like our supercompiler to terminate on all inputs — therefore, a termination test is required to control the amount of evaluation we perform. We would like to evaluate as much as possible (so the test should be lenient). Equally, if evaluation appears to start looping without achieving any simplification, then we would like to stop evaluating promptly (so the test should be vigilant).

Clearly, any test of this form will prevent us reducing some genuinely terminating terms to normal form (due to the Halting Problem), so all we can hope for is an approximation which does well in practice.

Concretely, let us say that we have a small-step evaluator for some language:

```
step :: Exp → Maybe Exp
```

The small-step evaluator is a partial function because some terms are already in normal form, and hence are irreducible. Given this small-step semantics we wish to define a big step semantics that evaluates an *Exp* to normal form:

```
reduce :: Exp → Exp
```

We would like *reduce* to be guaranteed to execute in finite time. How can we build such a function for a language for which strong normalisation does not hold? Clearly, we cannot, because many terms will never reduce to a normal form even if *step*ped an infinite number of times. To work around this problem, supercompilers relax the constraints on *reduce*: instead of returning a normal form, we would like *reduce* to return a normal form, *except when it looks like we will never reach one*.

Assuming a well-quasi-order *test* :: *TTest Exp* It is easy to define *reduce*:

```
reduce = go (initHistory test)
  where
    go hist s = case hist ‘test‘ s of
      Continue hist′ | Just s′ ← step s → go hist′ s′
      _                                 → s
```

The choice of the *test* well-quasi-order is what determines which heuristic is used for termination. The following three sections demonstrate how our combinators can capture the two most popular choices of termination test: the homeomorphic embedding on syntax trees (used in e.g. Klyuchnikov [8], Jonsson and Nordlander [9], Hamilton [10]), and the tag-bag well-quasi-order (used in e.g. Mitchell [2], Bolingbroke and Peyton Jones [11]).

7.2 Homeomorphic embedding on syntax trees

The homeomorphic embedding — previous alluded to in Section 5.1 — is a particular relation between (finite) labelled rose trees. The proof that it does indeed define a well-quasi-order is the famous “Tree Theorem” of Kruskal [1]. We can define it straightforwardly for the *Tree* type using our *gfixT* combinator:

```
type Tree a = Fix (TreeF a)
data TreeF a rec = NodeF a [rec]
                   deriving (Functor, Foldable, Traversable)
node :: a → [Tree a] → Tree a
node x ys = Roll (NodeF x ys)
treeT :: ∀a. TTest a → TTest (Tree a)
treeT wqo_elt = gfixT wqo_fix
  where
    wqo_fix :: ∀rec. TTest rec → TTest (TreeF a rec)
    wqo_fix wqo_subtree
      = cofmap inject (pairT wqo_elt (listT wqo_subtree))
    inject :: ∀rec. TreeF a rec → (a, [rec])
    inject (NodeF x ts) = (x, ts)
```

Now we have *treeT* — the homeomorphic embedding on rose trees — we can straightforwardly reuse it to define a homeomorphic embedding on *syntax* trees. To show how this test can be captured, we first define a simple data type of expressions, *Exp*:

```
data FnName = Map | Foldr | Even
              deriving (Enum, Bounded, Eq)

instance Finite FnName where
  elements = [minBound .. maxBound]
data Exp = FnVar FnName | Var String
         | App Exp Exp | Lam String Exp
         | Let String Exp Exp
```

As is standard, we identify a finite set of function names *FnName* that occur in the program to be supercompiled, distinct from the set of variables bound by lambads or **let**s. The purpose of this distinction is that we usually wish that $\neg(map \trianglelefteq foldr)$ but (since we assume an infinite supply of bound variables) we need that $x \trianglelefteq y$ within $\lambda x \to x \trianglelefteq \lambda y \to y$.

Our goal is to define a termination test *test1* :: *TTest Exp*. We proceed as follows:

```
data Node = FnVarN FnName | VarN
          | AppN | LamN | LetN
          deriving (Eq)
instance Finite Node where
  elements = VarN : AppN : LamN : LetN :
             map FnVarN elements
test1 :: TTest Exp
test1 = cofmap inject (treeT finiteT)
  where
    inject (FnVar x)    = node (FnVarN x) [ ]
    inject (Var _)      = node VarN [ ]
    inject (App e1 e2)  = node AppN [inject e1, inject e2]
    inject (Lam _ e)    = node LamN [inject e]
    inject (Let _ e1 e2) = node LetN [inject e1, inject e2]
```

The correctness of the *Finite Node* predicate is easy to verify, and thus this termination test is indeed a WQO. This test captures the standard use of the homeomorphic embedding in supercompilation.

More typically, the *FnName* data type will be a string, and the supercompiler will ensure that in any one execution of the supercompiler only a finite number of strings (the function names defined at the top level of the program to supercompile) will be placed into a *FnVar* constructor. In this case, the code for the termination test remains unchanged — but it is up to the supercompiler programmer to ensure that the new **instance** *Finite Node* declaration is justified.

7.3 Quasi-ordering tagged syntax trees

Observing that typical supercompiler implementations spent most of their time testing the termination criteria, Mitchell [2] proposed a simpler termination test based on “tag bags”. Our combinators are sufficient to capture this test, as we will demonstrate.

The idea of tags is that the syntax tree of the initial program has every node tagged with a unique number. As supercompilation proceeds, new syntax trees derived from the input syntax tree are created. This new syntax tree contains tags that may be copied and moved relative to their position in the original tree — but crucially the supercompiler will never tag a node with a totally new tag that comes "out of thin air". This property means that in any one run of the supercompiler we can assume that there are a finite number of tags.

We first require a type for these tags, for which we reuse Haskell's *Int* type. Crucially, *Int* is a bounded integer type (unlike *Integer*), so we can safely make the claim that *Tag* is *Finite*:

```
newtype Tag = Tag {unTag :: Int} deriving (Eq, Ord)
instance Finite Tag where
  elements = map Tag [minBound .. maxBound]
```

As there are rather a lot of distinct *Int*s, the well-quasi-order *finiteT* :: *TTest Tag* may potentially not reject sequences until they become very large indeed (i.e. it is not very vigilant). In practice, we will only have as many *Int* tags as we have nodes in the input program. Furthermore, most term sequences observed during supercompilation only use a fraction of these possible tags. For these reasons, these long sequences are never a problem in practice.

Continuing, we define the type of syntax trees where each node in the tree has a tag:

```
type TaggedExp = (Tag, TaggedExp′)
data TaggedExp′
  = TFnVar FnName | TVar String
  | TApp TaggedExp TaggedExp
  | TLam String TaggedExp
  | TLet String TaggedExp TaggedExp
```

We also need some utility functions for gathering all the tags from a tagged expression. There are many different subsets of the tags that you may choose to gather — one particular choice that closely follows Mitchell is as follows:

```
type TagBag = Map Tag Int
gather :: TaggedExp → TagBag
gather = go False
  where
    go lazy (tg, e) = singleton tg 1 ‘plus‘ go′ lazy e
    go′ lazy (TFnVar _)      = empty
    go′ lazy (TVar _)        = empty
    go′ lazy (TApp e1 e2)    = go lazy e1 ‘plus‘ go_lazy lazy e2
    go′ lazy (TLam _ e)      = empty
    go′ lazy (TLet _ e1 e2) = go_lazy lazy e1 ‘plus‘ go lazy e2
    go_lazy True  (tg, _) = singleton tg 1
    go_lazy False e       = go True e
    plus :: TagBag → TagBag → TagBag
    plus = unionWith (+)
```

We have assumed the following interface for constructing finite maps, with the standard meaning:

```
unionWith :: Ord k ⇒ (v → v → v)
             → Map k v → Map k v → Map k v
empty     :: Ord k ⇒ Map k v
singleton :: Ord k ⇒ k → v → Map k v
```

We can now define the tag-bag termination test of Mitchell [2] itself, *test2*:

```
test2 :: TTest TaggedExp
test2 = cofmap (summarise ∘ gather)
               (pairT finiteT wellOrderedT)
  where
    summarise :: TagBag → (Set Tag, Int)
    summarise tagbag
      = (keysSet tagbag, sum (elems tagbag))
```

7.4 Improved tag bags for tagged syntax trees

In fact, there is a variant of the tag-bag termination test that is more lenient than that of Mitchell [2]. Observe that the tag bag test as defined above causes the supercompiler to terminate when the domain of the tag bag is equal to a prior one and where the total number of elements in the bag has not decreased. However, since there are a finite number of tags, we can think of a tag-bag as simply a very large (but bounded) arity tuple — so by the Cartesian Product Lemma we need only terminate if *each of the tags considered individually* occur a non-decreasing number of times.

Our more lenient variant of the test can be defined in terms of the *finiteMapT* combinator of Section 4.7 almost trivially by reusing *gather*. It is straightforward to verify that if the *finiteMap* well-quasi-order relates two maps, those maps have exactly the same domains — so one of the parts of the original tag-bag termination test just falls out:

```
test3 :: TTest TaggedExp
test3 = cofmap gather (finiteMapT wellOrderedT)
```

All three of these termination tests — *test1*, *test2* and *test3* — are sound by construction, and straightforward to define using our library.

8. Related work

Leuschel [3] articulated why well-quasi-orders (and not, say, mere well-orders) are a particularly attractive choice for solving the online termination problem.

Our combinators all correspond to well-known lemmas about well-quasi-orders. A more complete survey of these lemmas can be found in Gallier [12] or Kruskal [1].

Perhaps surprisingly, similar ideas as those used for testing termination in the supercompilation literature have appeared in one of the most successful and influential static analysis approaches for program termination: the work stemming from *transition invariants* [13] and the *Terminator* tool [14]. In our case, we test that no subsequence of an input sequence is contained in a WQO. In the Terminator literature, static analysis guarantees that the transitive closure of the transition relation of a program is contained in a union of well-founded relations. It can be shown that the Terminator condition is closely related to the product formation for WQOs and we are currently preparing an article to explain these connections. It would be interesting to determine if techniques developed independently for testing with WQOs and the terminator literature can be ported over from one to the other. Finally, the use of homeomorphic embedding is also present in the static analysis world, where it is used to statically detect the termination of higher-order functions [15].

9. Conclusions and further work

We have shown that a library-based approach to constructing well-quasi-orders is practical: a small combinator set captures many common well-quasi-orders. Furthermore, well-quasi-orders constructed with these combinators are correct by construction — although combinators such as *finiteT* are only correct if some (simple) assumptions hold. We hope that verifying these base assumptions will prove much easier for the programmer than verifying whether or not something is a well-quasi-order. Our early experience using these combinators in a supercompiler indicates that their performance is quite acceptable for practical use: termination testing takes up only a fraction of the runtime of our supercompiler.

Hiding the implementation of the library allows it to be transparently replaced implementations that cache per-element work (Section 6.2) or prune the history of items seen (Section 6.1).

It would be interesting to try to extend the combinator language to capture the "refined" homeomorphic embedding of Klyuchnikov [16] – it does not seem to be expressible with the current set of combinators.

Implementing the combinators in a dependently typed language, such as Agda [17], would allow us to make the library truly correct-by-construction, as we could require the user of the library to supply proofs of things that we currently just assume – such as the finiteness of types tagged by $Finite$, or the more expressive dependent type of the "calibrator" function argument for the recursive types construction, or the correctness of the instantiation of fixT using $Traversable$.

The combinator library as described could be encoded as an instance of data-type-generic programming [18], and implemented in a language that supports such a paradigm (such as Generic Haskell [19]). This would give users of the library the option to take a generic implementation of a termination test for their data type, supplying well-quasi-orders only for the type constants in their system. It may also be interesting to extend fixT to support mutually-recursive systems of data types. It seems likely that existing work in the area of data-type-generics [20] would solve this problem straightforwardly.

Acknowledgments The authors would like to thank John Hughes for first suggesting an optimisation along the lines of Section 6.1 and Jeremy Gibbons for extensive discussions of Traversable functor properties. We would also like to thank the anonymous reviewers for their thought-provoking comments.

This work was partly supported by a PhD studentship generously provided by Microsoft Research.

References

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

[2] Neil Mitchell. Rethinking supercompilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*. ACM, 2010.

[3] Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-65014-0. doi: 10.1007/3-540-49727-7_14. URL `http://www.springerlink.com/content/g93wxkkwpfvvmnmg/`.

[4] Valentin F. Turchin. The algorithm of generalization in the supercompiler. *Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, Partial Evaluation and Mixed Computation*, pages 531–549, 1988.

[5] 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.

[6] Simon Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Kevin Hall, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

[7] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19, 2009. doi: 10.1017/S0956796809007291.

[8] Ilya Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009. URL `http://library.keldysh.ru/preprint.asp?lg=e&id=2009-63`.

[9] Peter A. Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2009.

[10] G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 61–70, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: http://doi.acm.org/10.1145/1244381.1244391. URL `http://doi.acm.org/10.1145/1244381.1244391`.

[11] Max Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, September 2010.

[12] Jean H. Gallier. What's so special about Kruskal's theorem and the ordinal γ_0? a survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.

[13] 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. ISBN 0-7695-2192-4. doi: 10.1109/LICS.2004.50. URL `http://portal.acm.org/citation.cfm?id=1018438.1021840`.

[14] 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. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134029. URL `http://doi.acm.org/10.1145/1133981.1134029`.

[15] Neil D. Jones and Nina Bohr. Termination analysis of the untyped lamba-calculus. In *RTA*, pages 1–23, 2004.

[16] Ilya Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, Moscow, 2010. URL `http://pat.keldysh.ru/~ilya/preprints/HOSC15_en.pdf`.

[17] 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.

[18] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-76786-2_1`.

[19] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer Berlin / Heidelberg, 2003. URL `http://dx.doi.org/10.1007/978-3-540-45191-4_1`.

[20] A.R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. *ACM SIGPLAN Notices*, 44(9):233–244, 2009.