Chasing Bottoms
A Case Study in Program Verification in the Presence of Partial and Infinite Values

Nils Anders Danielsson and Patrik Jansson*

Computing Science, Chalmers University of Technology, Gothenburg, Sweden

Abstract. This work is a case study in program verification: We have written a simple parser and a corresponding pretty-printer in a non-strict functional programming language with lifted pairs and functions (Haskell). A natural aim is to prove that the programs are, in some sense, each other's inverses. The presence of partial and infinite values in the domains makes this exercise interesting, and having lifted types adds an extra spice to the task. We have tackled the problem in different ways, and this is a report on the merits of those approaches. More specifically, we first describe a method for testing properties of programs in the presence of partial and infinite values. By testing before proving we avoid wasting time trying to prove statements that are not valid. Then we prove that the programs we have written are in fact (more or less) inverses using first fixpoint induction and then the approximation lemma.

1 Introduction

Infinite values are commonly used in (non-strict) functional programs, often to improve modularity [5]. Partial values are seldom used explicitly, but they are still present in all non-trivial Haskell programs because of non-termination, pattern match failures, calls to the *error* function etc. Unfortunately, proofs about functional programs often ignore details related to partial and infinite values.

This text is a case study where we explore how one can go about testing and proving properties even in the presence of partial and infinite values. We use random testing (Sect. 5) and two proof methods, fixpoint induction (Sect. 7) and the approximation lemma (Sect. 8), both described in Gibbons' and Hutton's tutorial [4].

The programs that our case study revolves around are a simple pretty-printer and a corresponding parser. Jansson and Jeuring define several more complex (polytypic) pretty-printers and parsers and prove them correct for total, finite input [7]. The case study in this paper uses cut down versions of those programs

* This work is partially funded by the Swedish Foundation for Strategic Research as part of the research programme "Cover — Combining Verification Methods in Software Development."

(see Sect. 2) but proves a stronger statement. On some occasions we have been tempted to change the definitions of the programs to be able to formulate our proofs in a different way. We have not done that, since one part of our goal is to explore what it is like to prove properties about programs that have not been written with a proof in mind. We have transformed our programs into equivalent variants, though; note that this carries a proof obligation.

Before starting to prove something it is often useful to test the properties. That way one can avoid spending time trying to prove something which is not true anyway. However, testing partial and infinite values can be tricky. In Sect. 5 we describe two techniques for doing that. Infinite values can be tested with the aid of the approximation lemma, and for partial values we make use of a Haskell extension, implemented in several Haskell environments. (The first technique is a generalisation of another one, and the last technique is previously known.)

As indicated above the programming language used for all programs and properties is Haskell [12], a non-strict, pure functional language where all types are lifted. Since we are careful with all details there will necessarily be some Haskell-specific discussions below, but the main ideas should carry over to other similar languages. Some knowledge of Haskell is assumed of the reader, though.

We begin in Sect. 2 by defining the two programs that this case study focuses on. Section 3 discusses the computational model and in Sect. 4 we give idealised versions of the main properties that we want to prove. By implementing and testing the properties (in Sect. 5) we identify a flaw in one of the them and we give a new, refined version in Sect. 6. The proofs presented in Sects. 7 and 8 are discussed in the concluding Sect. 9.

2 Programs

The programs under consideration parse and pretty-print a simple binary tree data type T without any information in the nodes:

$$\textbf{data}\ T = L \mid B\ T\ T$$

The pretty-printer is really simple. It performs a preorder traversal of the tree, emitting a `'B'` for each branching point and an `'L'` for each leaf:

$$\begin{aligned}
&\mathit{pretty}' :: T \to \mathit{String}\\
&\mathit{pretty}'\ L \qquad\ \ = \texttt{"L"}\\
&\mathit{pretty}'\ (B\ l\ r) = \texttt{"B"} +\!\!+\ \mathit{pretty}'\ l +\!\!+\ \mathit{pretty}'\ r
\end{aligned}$$

The parser reconstructs a tree given a string of the kind produced by pretty'. Any remaining input is returned together with the tree:

$$\begin{aligned}
&\mathit{parse} :: \mathit{String} \to (T, \mathit{String})\\
&\mathit{parse}\ (\texttt{'L'} : \mathit{cs}) = (L, \mathit{cs})\\
&\mathit{parse}\ (\texttt{'B'} : \mathit{cs}) = (B\ l\ r, \mathit{cs}'')\\
&\quad \textbf{where}\ (l,\ \mathit{cs}')\ \ = \mathit{parse}\ \mathit{cs}\\
&\qquad\qquad\ \ (r, \mathit{cs}'') = \mathit{parse}\ \mathit{cs}'
\end{aligned}$$

We wrap up $pretty'$ so that the printer and the parser get symmetric types:

$$pretty :: (T, String) \to String$$
$$pretty\ (t, cs) = pretty'\ t + cs$$

These programs are obviously written in a very naive way. A real pretty-printer would not use a quadratic algorithm for printing trees and a real parser would use a proper mechanism for reporting parse failures. However, the programs have the right level of detail for our application; they are very straightforward without being trivial. The tree structure makes the recursion "nonlinear," and that is what makes these programs interesting.

3 Computational Model

Before we begin reasoning about the programs we should specify what our underlying computational model is. We use Haskell 98 [12], and it is common to reason about Haskell programs by using equational reasoning, assuming that a simple denotational semantics for the language exists. This is risky, though, since this method has not been formally verified to work; there is not even a formal semantics for the language to verify it against. (We should mention that some work has been done on the static semantics [3].)

Nevertheless we will follow this approach, taking some caveats into account (see below). Although our aim is to explore what a proof would look like when all issues related to partial and infinite values are considered, it may be that we have missed some subtle aspect of the Haskell semantics. We have experimented with different levels of detail and believe that the resolution of such issues most likely will not change the overall structure of the proofs, though. Even if we would reject the idea of a clean denotational semantics for Haskell and instead use Sands' improvement theory [13] based on an operational model, we still believe that the proof steps would be essentially the same.

Now on to the caveats. All types in Haskell are (by default) pointed and lifted; each type is a complete partial order with a distinct least element $\bot$ (bottom), and data constructors are not strict. For pairs this means that $\bot \neq (\bot, \bot)$, so we do not have surjective pairing. It is possible to use strictness annotations to construct types that are not lifted, e.g. the smash product of two types, for which $\bot = (\bot, \bot)$ but we still do not have surjective pairing. There is however no way to construct the ordinary cartesian product of two types.

One has to be careful when using pattern matching in conjunction with lifted types. The expression **let** $(a, b) = x$ **in** $g\ (a, b)$ is equivalent to $g\ x$ iff $x \neq \bot$ or $g\ (\bot, \bot) = g\ \bot$. The reason is that, if $x = \bot$, then in the first case g will still be applied to $(\bot, \bot)$, whereas in the second case g will be applied to $\bot$. Note here that the pattern matching in a **let** clause is not performed until the variables bound in the pattern are actually used. Hence **let** $(a, b) = \bot$ **in** (a, b) is equivalent to $(\bot, \bot)$, whereas $(\lambda(a, b) \to (a, b))\ \bot = \bot$.

The function type is also lifted; we can actually distinguish between $\bot :: a \to a$ and $\lambda x \to \bot :: a \to a$ by using seq, a function with the following semantics [12]:

$seq :: a \to b \to b$
$seq\ \bot\ b = \bot$
$seq\ a\ \ b = b$

(Here a is any value except for $\bot$.) In other words η-conversion is not valid for Haskell functions, so to verify that two functions are equal it is not enough to verify that they produce identical output when applied to identical input; we also have to verify that none (or both) of the functions are $\bot$. A consequence of the lack of η-conversion is that one of the monadic identity laws fails to hold for some standard *Monad* instances in Haskell, such as the state "monad." The existence of a polymorphic *seq* also weakens Haskell's parametricity properties [8], but that does not directly affect us because our functions are not polymorphic.

Another caveat, also related to *seq*, is that $f = \lambda \mathit{True}\ x \to x$ is not identical to $f' = \lambda \mathit{True} \to \lambda x \to x$. By careful inspection of Haskell's pattern matching semantics [12] we can see that $f\ \mathit{False} = \lambda x \to \bot$ while $f'\ \mathit{False} = \bot$, since the function f is interpreted as

$\lambda a \to \lambda b \to$ **case** (a, b) **of**
  $(\mathit{True}, x) \to x$

whereas the function f' is interpreted as

$\lambda a \to$ **case** a **of**
  $\mathit{True} \to \lambda x \to x$.

This also applies if f and f' are defined by $f\ \ \mathit{True}\ x = x$ and $f'\ \ \mathit{True} = \lambda x \to x$. We do not get any problems if the first pattern is a simple variable, though. We will avoid problems related to this issue by never pattern matching on anything but the last variable in a multiple parameter function definition.

4 Properties: First Try

The programs in Sect. 2 are simple enough. Are they correct? That depends on what we demand of them. Let us say that we want them to form an embedding-projection pair, i.e.

$$\mathit{parse} \circ \mathit{pretty} = \mathit{id} :: (T, \mathit{String}) \to (T, \mathit{String}) \tag{1}$$

and

$$\mathit{pretty} \circ \mathit{parse} \sqsubseteq \mathit{id} :: \mathit{String} \to \mathit{String}. \tag{2}$$

The operator $\sqsubseteq$ denotes the ordering of the semantical domain, and $=$ is semantical equality.

More concretely (1) means that for all pairs $p :: (T, \mathit{String})$ we must have $\mathit{parse}\ (\mathit{pretty}\ p) = p$. (Note that η-conversion is valid since none of the functions involved are equal to $\bot$; they both expect at least one argument.) The quantification is over all pairs of the proper type, including infinite and partial values.

If we can prove this equality, then we are free to exchange the left and right hand sides in any well-typed context. This means that we can use the result very easily, but we have to pay a price in the complexity of the proof. In this section we "cheat" by only quantifying over finite, total trees so that we can use simple structural induction. We return to the full quantification in later sections.

Parse after Pretty. Let us prove (1) for finite, total trees and arbitrary strings, just to illustrate what this kind of proof usually looks like. First we observe that both sides are distinct from $\bot$, and then we continue using structural induction. The inductive hypothesis used is

$$\forall cs :: String\ .\ \ (parse \circ pretty)\ (t, cs) = id\ (t, cs),$$

where $t :: T$ is any immediate subtree of the tree treated in the current case. We have two cases, for the two constructors of T. The first case is easy (for an arbitrary $cs :: String$):

$$\begin{aligned}
&\ (parse \circ pretty)\ (L, cs)\\
=&\ \{\circ\}\\
&\ parse\ (pretty\ (L, cs))\\
=&\ \{pretty\}\\
&\ parse\ (pretty'\ L \mathbin{+\!\!+} cs)\\
=&\ \{pretty'\}\\
&\ parse\ (\texttt{"L"} \mathbin{+\!\!+} cs)\\
=&\ \{\mathbin{+\!\!+}\}\\
&\ parse\ (\texttt{'L'} : cs)\\
=&\ \{parse\}\\
&\ (L, cs)
\end{aligned}$$

The second case requires somewhat more work, but is still straightforward. (The use of **where** here is not syntactically correct, but is used for stylistic reasons. Just think of it as a postfix **let**.)

$$\begin{aligned}
&\ (parse \circ pretty)\ (B\ l\ r, cs)\\
=&\ \{\circ,\ pretty\}\\
&\ parse\ (pretty'\ (B\ l\ r) \mathbin{+\!\!+} cs)\\
=&\ \{pretty',\ \mathbin{+\!\!+}\ \text{associative},\ \mathbin{+\!\!+}\}\\
&\ parse\ (\texttt{'B'} : pretty'\ l \mathbin{+\!\!+} pretty'\ r \mathbin{+\!\!+} cs)\\
=&\ \{parse\}\\
&\ (B\ l'\ r', cs'')\\
&\quad \textbf{where}\ (l', cs')\ \ = parse\ (pretty'\ l \mathbin{+\!\!+} pretty'\ r \mathbin{+\!\!+} cs)\\
&\qquad\qquad\ \ (r', cs'') = parse\ cs'\\
=&\ \{pretty,\ \circ\}\\
&\ (B\ l'\ r', cs'')\\
&\quad \textbf{where}\ (l', cs')\ \ = (parse \circ pretty)\ (l, pretty'\ r \mathbin{+\!\!+} cs)\\
&\qquad\qquad\ \ (r', cs'') = parse\ cs'\\
=&\ \{\text{Inductive hypothesis}\}
\end{aligned}$$

$$
\begin{array}{l}
\quad (B\ l'\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (l', \mathit{cs}') \ \ = \mathit{id}\ (l, \mathit{pretty}'\ r \mathbin{+\!\!+} \mathit{cs}) \\
\qquad\qquad\qquad (r', \mathit{cs}'') = \mathit{parse}\ \mathit{cs}' \\
= \{\mathit{id},\ \textbf{where}\} \\
\quad (B\ l\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (r', \mathit{cs}'') = \mathit{parse}\ (\mathit{pretty}'\ r \mathbin{+\!\!+} \mathit{cs}) \\
= \{\mathit{pretty},\ \circ\} \\
\quad (B\ l\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (r', \mathit{cs}'') = (\mathit{parse} \circ \mathit{pretty})\ (r, \mathit{cs}) \\
= \{\text{Inductive hypothesis}\} \\
\quad (B\ l\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (r', \mathit{cs}'') = \mathit{id}\ (r, \mathit{cs}) \\
= \{\mathit{id},\ \textbf{where}\} \\
\quad (B\ l\ r, \mathit{cs})
\end{array}
$$

Hence we have proved using structural induction that $(\mathit{parse} \circ \mathit{pretty})\ (t, \mathit{cs}) = (t, \mathit{cs})$ for all finite, total $t :: T$ and for all $\mathit{cs} :: \mathit{String}$. Thus we can draw the conclusion that (1) is satisfied for that kind of input.

Pretty after Parse. We can show that (2) is satisfied in a similar way, using the fact that all Haskell functions are continuous and hence monotone with respect to $\sqsubseteq$. In fact, the proof works for arbitrary partial, finite input. We show the case for $\mathit{cs} :: \mathit{String}$, $\mathit{head}\ \mathit{cs} = \texttt{'B'}$, i.e. $\mathit{cs} = \texttt{'B'} : \mathit{cs}_1$ for some (partial and finite) $\mathit{cs}_1 :: \mathit{String}$:

$$
\begin{array}{l}
\quad (\mathit{pretty} \circ \mathit{parse})\ (\texttt{'B'} : \mathit{cs}_1) \\
= \{\circ,\ \mathit{parse}\} \\
\quad \mathit{pretty}\ (B\ l\ r, \mathit{cs}_1'') \\
\qquad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1 \\
\qquad\qquad\qquad (r, \mathit{cs}_1'') = \mathit{parse}\ \mathit{cs}_1' \\
= \{\mathit{pretty},\ \mathit{pretty}',\ \mathbin{+\!\!+}\ \text{associative}\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} \mathit{pretty}'\ l \mathbin{+\!\!+} \mathit{pretty}'\ r \mathbin{+\!\!+} \mathit{cs}_1'' \\
\qquad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1 \\
\qquad\qquad\qquad (r, \mathit{cs}_1'') = \mathit{parse}\ \mathit{cs}_1' \\
= \{\mathit{pretty}\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} \mathit{pretty}'\ l \mathbin{+\!\!+} \mathit{pretty}\ (r, \mathit{cs}_1'') \\
\qquad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1 \\
\qquad\qquad\qquad (r, \mathit{cs}_1'') = \mathit{parse}\ \mathit{cs}_1' \\
= \{\textbf{where},\ \mathit{pretty}\ \bot = \mathit{pretty}\ (\bot, \bot),\ \circ\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} \mathit{pretty}'\ l \mathbin{+\!\!+} (\mathit{pretty} \circ \mathit{parse})\ \mathit{cs}_1' \\
\qquad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1 \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} \mathit{pretty}'\ l \mathbin{+\!\!+} \mathit{id}\ \mathit{cs}_1' \\
\qquad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1 \\
= \{\mathit{id},\ \mathit{pretty},\ \textbf{where},\ \mathit{pretty}\ \bot = \mathit{pretty}\ (\bot, \bot),\ \circ\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} (\mathit{pretty} \circ \mathit{parse})\ \mathit{cs}_1 \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \texttt{"B"} \mathbin{+\!\!+} \mathit{id}\ \mathit{cs}_1 \\
= \{\mathit{id},\ \mathbin{+\!\!+}\} \\
\quad \texttt{'B'} : \mathit{cs}_1
\end{array}
$$

The other cases ($\mathit{head}\ \mathit{cs} \notin \{\texttt{'L'}, \texttt{'B'}\}$ and $\mathit{head}\ \mathit{cs} = \texttt{'L'}$) are both straightforward.

Parse after Pretty, Revisited. If we try to allow partial input in (1) instead of only total input, then we run into problems, as this counterexample shows:

$$
\begin{aligned}
& (\mathit{parse} \circ \mathit{pretty})\ (\bot, \mathit{cs}) \\
=\ & \{\circ,\ \mathit{pretty}\} \\
& \mathit{parse}\ (\mathit{pretty}'\ \bot \mathbin{+\!\!+} \mathit{cs}) \\
=\ & \{\mathit{pretty}',\ \mathbin{+\!\!+}\} \\
& \mathit{parse}\ \bot \\
=\ & \{\mathit{parse}\} \\
& \bot :: (T, \mathit{String}) \\
\neq\ & \{(,)\ \text{is not strict}\} \\
& (\bot, \mathit{cs}) :: (T, \mathit{String})
\end{aligned}
$$

We summarise our results so far in a table; we have proved (2) for finite, partial input and (1) for finite, total input. We have also disproved (1) in the case of partial input. The case marked with ? is treated in Sect. 5 below.

	Total	Partial
Finite	(2), (1)	(2), ¬ (1)
Infinite	?	¬ (1)

Hence the programs are not correct if we take (1) and (2) plus the type signatures of *pretty* and *parse* as our specification. Instead of refining the programs to meet this specification we will try to refine the specification. This approach is in line with our goal from Sect. 1: To prove properties of programs, without changing them.

5 Tests

As seen above we have to refine our properties, at least (1). To aid us in finding properties which are valid for partial and infinite input we will test the properties before we try to prove them.

How do we test infinite input in finite time? An approach which seems to work fine is to use the approximation lemma [6]. For T the function *approx* is defined as follows (*Nat* is a data type for natural numbers):

```
data Nat = Zero | Succ Nat
approx :: Nat → T → T
approx (Succ n) = λt → case t of
  L     → L
  B l r → B (approx n l) (approx n r)
```

Note that *approx Zero* is undefined, i.e. $\bot$. Hence *approx* n t traverses n levels down into the tree t and replaces everything there by $\bot$.

For the special case of trees the approximation lemma states that, for any $t_1, t_2 :: T$,

$$t_1 = t_2 \quad \text{iff} \quad \forall n \in \mathit{Nat}_{\mathrm{fin}}\,.\ \ \mathit{approx}\ n\ t_1 = \mathit{approx}\ n\ t_2. \tag{3}$$

Here Nat_{fin} stands for the total and finite values of type *Nat*, i.e. Nat_{fin} corresponds directly to $\mathbb{N}$. If we want to test that two expressions yielding possibly infinite trees are equal then we can use the right hand side of this equivalence. Of course we cannot test the equality for all n, but if it is not valid then running the test for small values of n should often be enough to find a counterexample.

Testing equality between lists using $take :: Int \to [a] \to [a]$ and the take lemma, an analogue to the approximation lemma, is relatively common. However, the former does not generalise as easily to other data types as the latter does. The approximation lemma generalises to any type which can be defined as the least fixpoint of a locally continuous functor [6]. This includes not only all polynomial types, but also much more, like nested and exponential types.

Using the approximation lemma we have now reduced testing of infinite values to testing of partial values. Thus even if we were dealing with total values only, we would still need to include $\perp$ in our tests. Generating the value $\perp$ is easily accomplished:

$$\begin{aligned}&\perp :: a\\&\perp = error\ \texttt{"\_|\_"}\end{aligned}$$

(Note that the same notation is used for the expression that generates a $\perp$ as for the value itself.)

The tricky part is testing for equality. If we do not want to use a separate tool then we necessarily have to use some impure extension, e.g. exception handling [11]. Furthermore it would be nice if we could perform these tests in pure code, such as QuickCheck [2] properties (see below). This can only be accomplished by using the decidedly unsafe function $unsafePerformIO :: IO\ a \to a$ [1, 11]. The resulting function $isBottom :: a \to Bool$[1] has to be used with care; it only detects a $\perp$ that results in an exception. However, that is enough for our purposes, since pattern match failures, *error* `"..."` and *undefined* all raise exceptions. If *isBottom x* terminates properly, then we can be certain that the answer produced (*True* or *False*) is correct.

Using *isBottom* we define a function that compares two arbitrary finite trees for equality:

```
(≙) :: T → T → Bool
t₁ ≙ t₂ = case (isBottom t₁, isBottom t₂) of
            (True, True)   → True
            (False, False) → case (t₁, t₂) of
              (L, L)             → True
              (B l r, B l′ r′)   → l ≙ l′ ∧ r ≙ r′
              _                  → False
            _ → False
```

[1] The function *isBottom* used here is a slight variation on the version implemented by Andy Gill in the libraries shipped with the GHC Haskell compiler. We have to take care not to catch e.g. stack overflow exceptions, as these may or may not correspond to bottoms.

Similarly we can define a function $(\hat{\sqsubseteq}) :: T \to T \to Bool$ which implements an approximation of the semantical domain ordering $(\sqsubseteq)$. The functions *approx*, $(\hat{=})$ and $(\hat{\sqsubseteq})$ are prime candidates for generalisation. We have implemented them using type classes; instances are generated automatically using the "Scrap Your Boilerplate" approach to generic programming [9].

QuickCheck is a library for defining and testing properties of Haskell functions [2]. By using the framework developed above we can now give QuickCheck implementations of properties (1) and (2):

$$
\begin{aligned}
&\mathit{prop}_1\ n = \mathit{forAll}\ \mathit{pair}\ (\lambda p \to \\
&\quad \mathit{approxPair}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ p)\ \hat{=}\ \mathit{approxPair}\ n\ (\mathit{id}\ p)) \\
&\mathit{prop}_2\ n = \mathit{forAll}\ \mathit{string}\ (\lambda \mathit{cs} \to \\
&\quad \mathit{approx}\ n\ ((\mathit{pretty} \circ \mathit{parse})\ \mathit{cs})\ \hat{\sqsubseteq}\ \mathit{approx}\ n\ (\mathit{id}\ \mathit{cs})) \\
&\mathit{approxPair}\ n\ (t, \mathit{cs}) = (\mathit{approx}\ n\ t, \mathit{approx}\ (2\,\hat{}\ n)\ \mathit{cs})
\end{aligned}
$$

These properties can be read more or less as ordinary set theoretic predicates, e.g. for prop_1 "for all pairs p the equality ... holds." The generators *pair* and *string* (defined in Appendix A) ensure that many different finite and infinite partial values are used for p and cs in the tests. Some values are never generated, though; see the end of this section.

If we run these tests then we see that prop_1 fails almost immediately, whereas prop_2 succeeds all the time. In other words (1) is not satisfied (which we already knew, see Sect. 4), but on the other hand we can be relatively certain that (2) is valid.

You might be interested in knowing whether (1) holds for *total* infinite input, a case which we have neglected above. We can easily write a test for such a case:

$$
\begin{aligned}
&\mathit{infiniteTree} = B\ \mathit{infiniteTree}\ L \\
&\mathit{propInfiniteTotal}\ n = \\
&\quad \mathit{approxPair}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ p)\ \hat{=}\ \mathit{approxPair}\ n\ (\mathit{id}\ p) \\
&\qquad \mathbf{where}\ p = (\mathit{infiniteTree}, \texttt{""})
\end{aligned}
$$

(The value *infiniteTree* is a left-infinite tree.) When executing this test we run into trouble, though; the test does not terminate for any $n \in \mathit{Nat}_{\mathrm{fin}}$. The reason is that the left-hand side does not terminate, and no part of the second component of the output pair is ever created (i.e. it is $\bot$). This can be seen by unfolding the expression a few steps:

$$
\begin{aligned}
&\quad \mathit{approxPair}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ (\mathit{infiniteTree}, \texttt{""})) \\
&= \{\text{Unfold, rearrange slightly}\} \\
&\quad (\mathit{approx}\ n\ (B\ l\ r), \mathit{approx}\ (2\,\hat{}\ n)\ \mathit{cs}'') \\
&\qquad \mathbf{where}\ (l, \mathit{cs}') \ = (\mathit{parse} \circ \mathit{pretty})\ (\mathit{infiniteTree}, \texttt{"L"}) \\
&\qquad\qquad\quad (r, \mathit{cs}'') = \mathit{parse}\ \mathit{cs}'
\end{aligned}
$$

One of the subexpressions is $(\mathit{parse} \circ \mathit{pretty})\ (\mathit{infiniteTree}, \texttt{"L"})$, which is essentially the same expression as the one that we started out with, and cs'' will not be generated until that subexpression has produced any output in its second

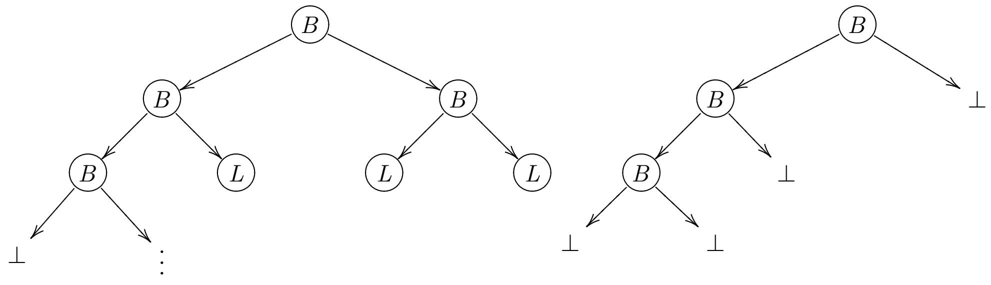

Fig. 1: With the left tree called t, the right tree is $t' = (\mathit{fst} \circ \mathit{parse} \circ \mathit{pretty}')\ t$.

component. The right-hand side does terminate, though, so (1) is not valid for total, infinite input.

Since prop_1 does not terminate for total, infinite trees we have designed our QuickCheck generators so that they do not generate such values. This is of course a slight drawback.

6 Properties: Second Try

As noted above (1) is not valid in general. If we inspect what happens when $\mathit{fst} \circ \mathit{parse} \circ \mathit{pretty}'$ is applied to a partial tree, then we see that as soon as a $\perp$ is encountered all nodes encountered later in a preorder traversal of the tree are replaced by $\perp$ (see Fig. 1).

We can easily verify that the example in the figure is correct (assuming that the part represented by the vertical dots is a left-infinite total tree):

$$\begin{aligned}
t &= B\ (B\ (B\ \perp\ \mathit{infiniteTree})\ L)\ (B\ L\ L)\\
t' &= B\ (B\ (B\ \perp\ \perp\qquad\qquad\ )\ \perp)\ \perp\\
\mathit{propFigure} &= t' \mathrel{\hat{=}} (\mathit{fst} \circ \mathit{parse} \circ \mathit{pretty}')\ t
\end{aligned}$$

Evaluating *propFigure* yields *True*, as expected.

Given this background it is not hard to see that $(\mathit{snd} \circ \mathit{parse} \circ \mathit{pretty})\ (t, \mathit{cs}) = \perp$ whenever the tree t is not total. Furthermore $(\mathit{parse} \circ \mathit{pretty})\ (t, \mathit{cs}) = \perp$ iff $t = \perp$. Using the preceding results we can write a replacement *strictify* for *id* that makes

$$\mathit{parse} \circ \mathit{pretty} = \mathit{strictify} :: (T, \mathit{String}) \rightarrow (T, \mathit{String}) \tag{1'}$$

a valid refinement of (1) (as we will see below):

$$\begin{aligned}
&\mathit{strictify} :: (T, a) \rightarrow (T, a)\\
&\mathit{strictify}\ (t, a) = t\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ a)\\
&\quad \mathbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ t
\end{aligned}$$

If $t = \bot$ then $\bot$ should be returned, hence the first *seq*. The helper function *strictify′*, which does the main trunk of the work, returns the strictified tree in its first component. The second component, which is threaded bottom-up through the computation, is () whenever the input tree is total, and $\bot$ otherwise; hence the second *seq*. In effect we use the Haskell type () as a boolean type with $\bot$ as falsity and () as truth. It is the threading of this "boolean," in conjunction with the sequential nature of *seq*, which enforces the preorder traversal and strictification indicated in the figure above:

```
strictify' :: T → (T, ())
strictify' L       = (L, ())
strictify' (B l r) = (B l' (lTotal `seq` r'), lTotal `seq` rTotal)
  where (l', lTotal) = strictify' l
        (r', rTotal) = strictify' r
```

Note that if the left subtree l is not total, then the right subtree r should be replaced by $\bot$; hence the use of *lTotal* ‘*seq*‘ r' above. The second component should be () iff both subtrees are total, so we use *seq* as logical and between *lTotal* and *rTotal*; a ‘*seq*‘ b = () iff a = () and b = () for a, b :: ().

Before we go on to prove (1'), let us test it:

```
prop'_1 n = forAll pair (λp →
  approxPair n ((parse ∘ pretty) p) ≙ approxPair n (strictify p))
```

This test seems to succeed all the time — a good indication that we are on the right track.

7 Proofs Using Fixpoint Induction

Now we will prove (1') and (2) using two different methods, fixpoint induction (in this section) and the approximation lemma (in Sect. 8). All details will not be presented, since that would take up too much space.

In this section let ψ, ψ_i etc. stand for arbitrary types.

To be able to use fixpoint induction [4, 14] all recursive functions have to be defined using *fix*, which is defined by

$$\mathit{fix}\, f = \bigsqcup_{i=0}^{\infty} f^i \bot \tag{4}$$

for any continuous function $f :: \psi \to \psi$. (The notation f^i stands for f composed with itself i times.) It is easy to implement *fix* in Haskell, but proving that the two definitions are equivalent would take up too much space, and is omitted:

```
fix :: (a → a) → a
fix f = f (fix f)
```

Let P be a chain-complete predicate, i.e. a predicate which is true for the least upper bound of a chain whenever it is true for all the elements in the chain. In other words, if $P(f^i \bot)$ is true for all $i \in \mathbb{N}$ and some $f :: \psi \to \psi$, then we know that $P(\mathit{fix}\ f)$ is true (we only consider ω-chains). Generalising we get the following inference rule from ordinary induction over natural numbers (and some simple domain theory):

$$\frac{P(\bot, \bot, \ldots, \bot) \qquad \begin{array}{c}\forall n \in \mathbb{N}.\ (P(f_1^n \bot, f_2^n \bot, \ldots, f_m^n \bot) \Rightarrow \\ P(f_1^{n+1} \bot, f_2^{n+1} \bot, \ldots, f_m^{n+1} \bot))\end{array}}{P(\mathit{fix}\ f_1, \mathit{fix}\ f_2, \ldots, \mathit{fix}\ f_m)} \tag{5}$$

Here $m \in \mathbb{N}$ and the f_i are continuous functions $f_i :: \psi_i \to \psi_i$. We also have the following useful variant which follows immediately from the previous one, assuming that the ψ_i are function types, $\psi_i = \psi'_i \to \psi''_i$, and that all f_i are strictness-preserving, i.e. if g_i is strict then $f_i\ g_i$ should be strict as well.

$$\frac{P(\bot, \bot, \ldots, \bot) \quad \begin{array}{c}\forall \text{ strict } g_1 :: \psi_1, g_2 :: \psi_2, \ldots, g_m :: \psi_m\ . \\ P(g_1, g_2, \ldots, g_m) \Rightarrow P(f_1\ g_1, f_2\ g_2, \ldots, f_m\ g_m)\end{array}}{P(\mathit{fix}\ f_1, \mathit{fix}\ f_2, \ldots, \mathit{fix}\ f_m)} \tag{6}$$

That is all the theory that we need for now; on to the proofs. Let us begin by defining variants of our recursive functions using *fix*:

```
pretty'_fix :: T → String
pretty'_fix = fix pretty_step
pretty_step :: (T → String) → T → String
pretty_step p L       = "L"
pretty_step p (B l r) = "B" ++ p l ++ p r

parse_fix :: String → (T, String)
parse_fix = fix parse_step
parse_step :: (String → (T, String)) → String → (T, String)
parse_step p' ('L' : cs) = (L, cs)
parse_step p' ('B' : cs) = (B l r, cs'')
  where (l, cs')  = p' cs
        (r, cs'') = p' cs'

strictify'_fix :: T → (T, ())
strictify'_fix = fix strictify_step
strictify_step :: (T → (T, ())) → T → (T, ())
strictify_step s L       = (L, ())
strictify_step s (B l r) = (B l' (lTotal `seq` r'), lTotal `seq` rTotal)
  where (l', lTotal) = s l
        (r', rTotal) = s r
```

Of course using these definitions instead of the original ones implies a proof obligation; we have to show that the two sets of definitions are equivalent to each other. In a standard domain theoretic setting this would follow immediately from the interpretation of a recursively defined function. In the case of Haskell this requires some work, though. The proofs are certainly possible to perform, but they would lead us too far astray, so we omit them here.

The properties have to be unrolled to fit the requirements of the inference rules. To make the properties more readable we define new versions of some other functions as well:

$$\begin{aligned}
&\mathit{pretty}_{\mathit{fix}} :: (T \to \mathit{String}) \to (T, \mathit{String}) \to \mathit{String}\\
&\mathit{pretty}_{\mathit{fix}}\ p\ (t, \mathit{cs}) = p\ t +\!\!+\ \mathit{cs}\\
&\mathit{strictify}_{\mathit{fix}} :: (T \to (T, ())) \to (T, a) \to (T, a)\\
&\mathit{strictify}_{\mathit{fix}}\ s\ (t, a) = t\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ a)\\
&\quad \mathbf{where}\ (t', \mathit{tTotal}) = s\ t
\end{aligned}$$

We end up with

$$P_1(p, p', s) = \qquad p' \circ \mathit{pretty}_{\mathit{fix}}\ p = \mathit{strictify}_{\mathit{fix}}\ s \tag{7}$$

and

$$P_2(p, p') = \qquad \mathit{pretty}_{\mathit{fix}}\ p \circ p' \sqsubseteq \mathit{id}. \tag{8}$$

However, we cannot use P_1 as it stands since $P_1(\bot, \bot, \bot)$ is not true. To see this, pick an arbitrary $\mathit{cs} :: \mathit{String}$ and a $t :: T$ satisfying $t \neq \bot$:

$$\begin{aligned}
&\quad (\bot \circ \mathit{pretty}_{\mathit{fix}}\ \bot)\ (t, \mathit{cs})\\
&= \{\circ, \bot\}\\
&\quad \bot :: (T, \mathit{String})\\
&\neq \{\mathit{seq},\ t \neq \bot,\ (,)\text{ is not strict}\}\\
&\quad t\ \text{`}\mathit{seq}\text{`}\ (\bot, \bot) :: (T, \mathit{String})\\
&= \{\mathit{seq}\}\\
&\quad t\ \text{`}\mathit{seq}\text{`}\ (\bot, \bot\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) :: (T, \mathit{String})\\
&= \{\mathbf{where}, \text{pattern matching}\}\\
&\quad t\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) :: (T, \mathit{String})\\
&\qquad \mathbf{where}\ (t', \mathit{tTotal}) = \bot\\
&= \{\bot\}\\
&\quad t\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) :: (T, \mathit{String})\\
&\qquad \mathbf{where}\ (t', \mathit{tTotal}) = \bot\ t\\
&= \{\mathit{strictify}_{\mathit{fix}}\}\\
&\quad \mathit{strictify}_{\mathit{fix}}\ \bot\ (t, \mathit{cs})
\end{aligned}$$

We can still go on by noticing that we are only interested in the property in the limit and redefining it as

$$P_1'(p, p', s) = P_1(\mathit{pretty}_{\mathit{step}}\ p, \mathit{parse}_{\mathit{step}}\ p', \mathit{strictify}_{\mathit{step}}\ s), \tag{7'}$$

i.e. $P'_1(p, p', s)$ is equivalent to

$$\mathit{parse}_{\mathit{step}}\ p' \circ \mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p) = \mathit{strictify}_{\mathit{fix}}\ (\mathit{strictify}_{\mathit{step}}\ s). \tag{9}$$

With P'_1 we avoid the troublesome base case since $P'_1(\bot, \bot, \bot)$ is equivalent to $P_1(\mathit{pretty}_{\mathit{step}}\ \bot, \mathit{parse}_{\mathit{step}}\ \bot, \mathit{strictify}_{\mathit{step}}\ \bot)$.

Now it is straightforward to verify that $P'_1(\bot, \bot, \bot)$ and $P_2(\bot, \bot)$ are valid (P'_1 requires a tedious but straightforward case analysis). It is also easy to verify that the predicates are chain-complete using general results from domain theory [14]. As we have already stated above, verifying formally that $P'_1(\mathit{fix}\ \mathit{pretty}_{\mathit{step}}, \mathit{fix}\ \mathit{parse}_{\mathit{step}}, \mathit{fix}\ \mathit{strictify}_{\mathit{step}})$ is equivalent to (1') and similarly that $P_2(\mathit{fix}\ \mathit{pretty}_{\mathit{step}}, \mathit{fix}\ \mathit{parse}_{\mathit{step}})$ is equivalent to (2) requires more work and is omitted.

Pretty after Parse. Now on to the main work. Let us begin with P_2. Since we do not need the tighter inductive hypothesis of inference rule (5) we will use inference rule (6); it is easy to verify that $\mathit{pretty}_{\mathit{step}}$ and $\mathit{parse}_{\mathit{step}}$ are strictness-preserving. Assume now that $P_2(p, p')$ is valid for strict $p :: T \to \mathit{String}$ and $p' :: \mathit{String} \to (T, \mathit{String})$. We have to show that $P_2(\mathit{pretty}_{\mathit{step}}\ p, \mathit{parse}_{\mathit{step}}\ p')$ is valid. After noting that both sides of the inequality are distinct from $\bot$, take an arbitrary element $cs :: \mathit{String}$. The proof is a case analysis on $\mathit{head}\ cs$.

First case, $\mathit{head}\ cs \notin \{\texttt{'L'}, \texttt{'B'}\}$:

$$\begin{aligned}
&(\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p) \circ (\mathit{parse}_{\mathit{step}}\ p'))\ cs \\
=\ &\{\circ,\ \mathit{parse}_{\mathit{step}},\ \mathit{head}\ cs \notin \{\texttt{'L'}, \texttt{'B'}\}\} \\
&\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p)\ \bot \\
=\ &\{\mathit{pretty}_{\mathit{fix}}\} \\
&\bot :: \mathit{String} \\
\sqsubseteq\ &\{\bot \text{ is the least element in the domain}\} \\
&\mathit{id}\ cs
\end{aligned}$$

Second case, $\mathit{head}\ cs = \texttt{'L'}$, i.e. $cs = \texttt{'L'} : cs_1$ for some $cs_1 :: \mathit{String}$:

$$\begin{aligned}
&(\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p) \circ (\mathit{parse}_{\mathit{step}}\ p'))\ cs \\
=\ &\{\circ,\ \mathit{parse}_{\mathit{step}},\ cs = \texttt{'L'} : cs_1\} \\
&\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p)\ (L, cs_1) \\
=\ &\{\mathit{pretty}_{\mathit{fix}}\} \\
&\mathit{pretty}_{\mathit{step}}\ p\ L \mathbin{+\!\!+} cs_1 \\
=\ &\{\mathit{pretty}_{\mathit{step}},\ \mathbin{+\!\!+},\ \mathit{id}\} \\
&\mathit{id}\ cs
\end{aligned}$$

Last case, $\mathit{head}\ cs = \texttt{'B'}$, i.e. $cs = \texttt{'B'} : cs_1$ for some $cs_1 :: \mathit{String}$:

$$\begin{aligned}
&(\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p) \circ (\mathit{parse}_{\mathit{step}}\ p'))\ cs \\
=\ &\{\circ,\ \mathit{parse}_{\mathit{step}},\ cs = \texttt{'B'} : cs_1\} \\
&\mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p)\ (B\ l\ r, cs''_1) \\
&\quad \textbf{where}\ (l, cs'_1) = p'\ cs_1 \\
&\quad\phantom{\textbf{where}}\ (r, cs''_1) = p'\ cs'_1
\end{aligned}$$

$$
\begin{array}{l}
= \{\mathit{pretty}_{\mathit{fix}},\ \mathit{pretty}_{\mathit{step}},\ +\!\!+ \text{ associative}\} \\
\quad \texttt{"B"} +\!\!+ p\ l +\!\!+ (p\ r +\!\!+ cs_1'') \\
\qquad \textbf{where } (l,\ cs_1') = p'\ cs_1 \\
\qquad\qquad\quad\ (r,\ cs_1'') = p'\ cs_1' \\
= \{\mathit{pretty}_{\mathit{fix}}\} \\
\quad \texttt{"B"} +\!\!+ p\ l +\!\!+ \mathit{pretty}_{\mathit{fix}}\ p\ (r,\ cs_1'') \\
\qquad \textbf{where } (l,\ cs_1') = p'\ cs_1 \\
\qquad\qquad\quad\ (r,\ cs_1'') = p'\ cs_1' \\
= \{\textbf{where},\ p \text{ strict implies that } \mathit{pretty}_{\mathit{fix}}\ p\ \bot = \mathit{pretty}_{\mathit{fix}}\ p\ (\bot, \bot),\ \circ\} \\
\quad \texttt{"B"} +\!\!+ p\ l +\!\!+ (\mathit{pretty}_{\mathit{fix}}\ p \circ p')\ cs_1' \\
\qquad \textbf{where } (l, cs_1') = p'\ cs_1 \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \texttt{"B"} +\!\!+ p\ l +\!\!+ \mathit{id}\ cs_1' \\
\qquad \textbf{where } (l, cs_1') = p'\ cs_1 \\
= \{\mathit{id},\ \mathit{pretty}_{\mathit{fix}},\ \textbf{where},\ p \text{ strict},\ \circ\} \\
\quad \texttt{"B"} +\!\!+ (\mathit{pretty}_{\mathit{fix}}\ p \circ p')\ cs_1 \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \texttt{"B"} +\!\!+ \mathit{id}\ cs_1 \\
= \{\mathit{id},\ +\!\!+,\ \mathit{id}\} \\
\quad \mathit{id}\ cs
\end{array}
$$

This concludes the proof for P_2.

Parse after Pretty. For P_1' we will also use inference rule (6); in addition to $\mathit{pretty}_{\mathit{step}}$ and $\mathit{parse}_{\mathit{step}}$ it is easy to verify that $\mathit{strictify}_{\mathit{step}}$ is strictness-preserving. Assume that $P_1'(p_0, p_0', s_0)$ is valid, where p_0, p_0' and s_0 are all strict. We have to prove that $P_1'(\mathit{pretty}_{\mathit{step}}\ p_0,\ \mathit{parse}_{\mathit{step}}\ p_0', \mathit{strictify}_{\mathit{step}}\ s_0)$ is valid. This is equivalent to proving $P_1(\mathit{pretty}_{\mathit{step}}\ p,\ \mathit{parse}_{\mathit{step}}\ p',\ \mathit{strictify}_{\mathit{step}}\ s)$, where $p = \mathit{pretty}_{\mathit{step}}\ p_0$, $p' = \mathit{parse}_{\mathit{step}}\ p_0'$ and $s = \mathit{strictify}_{\mathit{step}}\ s_0$. The first step of this proof is to note that both sides of the equality in P_1' are distinct from $\bot$. The rest of the proof is, as before, performed using case analysis, this time on pair, an arbitrary element in (T, cs). The cases $\mathit{pair} = \bot$, $\mathit{pair} = (\bot, cs)$ and $\mathit{pair} = (L, cs)$ for an arbitrary $cs :: \mathit{String}$ are straightforward and omitted.

Last case, $\mathit{pair} = (B\ l\ r, cs)$ for arbitrary subtrees $l, r :: T$ and an arbitrary $cs :: \mathit{String}$:

$$
\begin{array}{l}
\quad (\mathit{parse}_{\mathit{step}}\ p' \circ \mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p))\ (B\ l\ r, cs) \\
= \{\circ,\ \mathit{pretty}_{\mathit{fix}}\} \\
\quad \mathit{parse}_{\mathit{step}}\ p'\ (\mathit{pretty}_{\mathit{step}}\ p\ (B\ l\ r) +\!\!+ cs) \\
= \{\mathit{pretty}_{\mathit{step}},\ +\!\!+,\ +\!\!+ \text{ associative}\} \\
\quad \mathit{parse}_{\mathit{step}}\ p'\ (\texttt{'B'} : p\ l +\!\!+ (p\ r +\!\!+ cs)) \\
= \{\mathit{parse}_{\mathit{step}}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where } (l',\ cs')\ = p'\ (p\ l +\!\!+ (p\ r +\!\!+ cs)) \\
\qquad\qquad\quad\ (r', cs'') = p'\ cs' \\
= \{\mathit{pretty}_{\mathit{fix}},\ \circ\}
\end{array}
$$

$$
\begin{array}{l}
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l',\ cs') \ = (p' \circ \mathit{pretty}_{\mathit{fix}}\ p)\ (l, p\ r \mathbin{+\!\!+} cs) \\
\qquad\qquad\quad (r', cs'') = p'\ cs' \\
= \{\text{Inductive hypothesis}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l',\ cs') \ = \mathit{strictify}_{\mathit{fix}}\ s\ (l, p\ r \mathbin{+\!\!+} cs) \\
\qquad\qquad\quad (r', cs'') = p'\ cs' \\
= \{\mathit{strictify}_{\mathit{fix}}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l',\ cs') \qquad = l\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs) \\
\qquad\qquad\quad (t', \mathit{tTotal}) = s\ l \\
\qquad\qquad\quad (r', cs'') \qquad = p'\ cs' \\
= \{\text{Simple case analysis on } l\ (\bot \text{ or not } \bot)\text{, pattern matching}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l',\ cs') \qquad = (l\ \text{`}\mathit{seq}\text{`}\ t', l\ \text{`}\mathit{seq}\text{`}\ \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs) \\
\qquad\qquad\quad (t', \mathit{tTotal}) = s\ l \\
\qquad\qquad\quad (r', cs'') \qquad = p'\ cs' \\
= \{\mathit{seq}\text{, if } l = \bot \text{ then } t' = \mathit{tTotal} = \bot \text{ since } s \text{ is strict}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l',\ cs') \qquad = (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs) \\
\qquad\qquad\quad (t', \mathit{tTotal}) = s\ l \\
\qquad\qquad\quad (r', cs'') \qquad = p'\ cs' \\
= \{\textbf{where}\} \\
\quad (B\ t'\ r', cs'') \\
\qquad \textbf{where}\ (t', \mathit{tTotal}) = s\ l \\
\qquad\qquad\quad (r', cs'') \qquad = p'\ (\mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs) \\
= \{\text{Rename variables}\} \\
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (l', \mathit{lTotal}) = s\ l \\
\qquad\qquad\quad (r', cs'') \qquad = p'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs)
\end{array}
$$

The rest of the proof is straightforward. Using case analysis on *lTotal* we prove that

$$
\begin{array}{l}
\quad (B\ l'\ r', cs'') \\
\qquad \textbf{where}\ (r', cs'') = p'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs) \\
= \\
\quad (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}\ \text{`}\mathit{seq}\text{`}\ cs) \\
\qquad \textbf{where}\ (r', \mathit{rTotal}) = s\ r
\end{array}
$$

is valid. In one branch one can observe that p' is strict. In the other the inductive hypothesis can be applied, followed by reasoning analogous to the one for l `*seq*` above. Given this equality the rest of the proof is easy. Hence we can draw the conclusion that $P_1(\mathit{pretty}_{\mathit{step}}\ p, \mathit{parse}_{\mathit{step}}\ p', \mathit{strictify}_{\mathit{step}}\ s)$ is valid, which means that we have finished the proof.

8 Proofs Using the Approximation Lemma

Let us now turn to the approximation lemma. This lemma was presented above in Sect. 5, but we still have a little work to do before we can go to the proofs.

Pretty after Parse. Any naive attempt to prove (2) using the obvious inductive hypothesis fails. Using the following less obvious reformulated property does the trick, though:

$$\forall m \in \mathbb{N}\,.\;\; pp_m \sqsubseteq id :: String \to String. \tag{10}$$

Here we use a family of helper functions pp_m ($m \in \mathbb{N}$):

$$\begin{aligned}
&pp_m\ cs = pretty'\ t_1 +\!\!+\ pretty'\ t_2 +\!\!+ \ldots +\!\!+\ pretty'\ t_m +\!\!+\ cs_m\\
&\quad \textbf{where}\ (t_1,\ cs_1)\ = parse\ cs\\
&\qquad\qquad\ (t_2,\ cs_2)\ = parse\ cs_1\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\ (t_m, cs_m) = parse\ cs_{m-1}
\end{aligned}$$

(We interpret pp_0 as id.) It is straightforward to verify that this property is equivalent to (2).

Note that we cannot use the approximation lemma directly as it stands, since the lemma deals with equalities, not inequalities. However, replacing each $=$ with $\sqsubseteq$ in the proof of the approximation lemma in Gibbons' and Hutton's article [4, Sect. 4] is enough to verify this variant. We get that, for all $m \in \mathbb{N}$ and $cs :: String$,

$$\begin{aligned}
pp_m\ cs \sqsubseteq id\ cs \quad \text{iff}\\
\forall n \in Nat_{\text{fin}}\,.\;\; approx\ n\ (pp_m\ cs) \sqsubseteq approx\ n\ (id\ cs).
\end{aligned} \tag{11}$$

Hence all that we need to do is to prove the last statement above (after noticing that both pp_m and id are distinct from $\bot$, for all $m \in \mathbb{N}$). We do that by induction over n, after observing that we can change the order of the universal quantifiers so that we get

$$\begin{aligned}
&\forall n \in Nat_{\text{fin}}\,.\;\; \forall m \in \mathbb{N}\,.\;\; \forall cs :: String\,.\\
&\qquad\qquad\qquad approx\ n\ (pp_m\ cs) \sqsubseteq approx\ n\ (id\ cs),
\end{aligned} \tag{12}$$

which is equivalent to the inequalities above.

For lists we have the following variant of *approx*:

$$\begin{aligned}
&approx :: Nat \to [\,a\,] \to [\,a\,]\\
&approx\ (Succ\ n) = \lambda(x : xs) \to x : approx\ n\ xs
\end{aligned}$$

Since *approx Zero* is undefined the statement (12) is trivially true for $n = Zero$. Assume now that $\forall m \in \mathbb{N}\,.\; \forall cs :: String\,.\; approx\ n\ (pp_m\ cs) \sqsubseteq approx\ n\ (id\ cs)$ is true for some $n \in Nat_{\text{fin}}$. Take an arbitrary $m \in \mathbb{N}$. Note that the property that we want to prove is trivially true for $m = 0$, so assume that $m \geq 1$. We proceed by case analysis on *head cs*.

First case, $head\ cs \notin \{$`'L'`, `'B'`$\}$:

$$\begin{aligned}
&\quad approx\ (Succ\ n)\ (pp_m\ cs)\\
&= \{parse,\ \textbf{where},\ pretty',\ +\!\!+\}\\
&\quad approx\ (Succ\ n)\ \bot\\
&\sqsubseteq \{\bot \text{ is the least element, monotonicity}\}\\
&\quad approx\ (Succ\ n)\ (id\ cs)
\end{aligned}$$

Second case, $\mathit{head}\ cs = \mathtt{'L'}$, i.e. $cs = \mathtt{'L'} : cs'$ for some $cs' :: \mathit{String}$:

$$
\begin{array}{l}
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (pp_m\ (\mathtt{'L'} : cs')) \\
= \{pp_m,\ m \geq 1\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ t_1 \mathbin{+\!\!+} \mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(t_1,\ cs_1) & = \mathit{parse}\ (\mathtt{'L'} : cs') \\
(t_2,\ cs_2) & = \mathit{parse}\ cs_1 \\
 & \vdots \\
(t_m, cs_m) & = \mathit{parse}\ cs_{m-1}
\end{array} \\
= \{\mathit{parse},\ \textbf{where}, \text{ note that if } m = 1 \text{ then } cs_m = cs'\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ L \mathbin{+\!\!+} \mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(t_2,\ cs_2) & = \mathit{parse}\ cs' \\
 & \vdots \\
(t_m, cs_m) & = \mathit{parse}\ cs_{m-1}
\end{array} \\
= \{\mathit{pretty}',\ \mathbin{+\!\!+}\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathtt{'L'} : \mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(t_2,\ cs_2) & = \mathit{parse}\ cs' \\
 & \vdots \\
(t_m, cs_m) & = \mathit{parse}\ cs_{m-1}
\end{array} \\
= \{\mathit{approx}\} \\
\quad \mathtt{'L'} : \mathit{approx}\ n\ (\mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(t_2,\ cs_2) & = \mathit{parse}\ cs' \\
 & \vdots \\
(t_m, cs_m) & = \mathit{parse}\ cs_{m-1}
\end{array} \\
= \big\{pp_{m-1},\ m \geq 1\big\} \\
\quad \mathtt{'L'} : \mathit{approx}\ n\ (pp_{m-1}\ cs') \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \mathtt{'L'} : \mathit{approx}\ n\ (\mathit{id}\ cs') \\
= \{\mathit{id},\ \mathit{approx}\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathtt{'L'} : cs')
\end{array}
$$

Last case, $\mathit{head}\ cs = \mathtt{'B'}$, i.e. $cs = \mathtt{'B'} : cs'$ for some $cs' :: \mathit{String}$:

$$
\begin{array}{l}
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (pp_m\ (\mathtt{'B'} : cs')) \\
= \{pp_m,\ m \geq 1\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ t_1 \mathbin{+\!\!+} \mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(t_1,\ cs_1) & = \mathit{parse}\ (\mathtt{'B'} : cs') \\
(t_2,\ cs_2) & = \mathit{parse}\ cs_1 \\
 & \vdots \\
(t_m, cs_m) & = \mathit{parse}\ cs_{m-1}
\end{array} \\
= \{\mathit{parse},\ \textbf{where}\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ (B\ l\ r) \mathbin{+\!\!+} \mathit{pretty}'\ t_2 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathit{pretty}'\ t_m \mathbin{+\!\!+} cs_m) \\
\qquad \textbf{where}\ \begin{array}[t]{ll}
(l,\ \ ls) & = \mathit{parse}\ cs' \\
(r,\ \ rs) & = \mathit{parse}\ ls \\
(t_2,\ cs_2) & = \mathit{parse}\ rs \\
 & \vdots
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\vdots \\
\qquad\qquad\quad (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
= \{\mathit{pretty}', +\!\!+, +\!\!+ \text{ associative}\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n) \\
\quad\quad (\texttt{'B'} : \mathit{pretty}'\ l +\!\!+ \mathit{pretty}'\ r +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ cs_m) \\
\quad\quad \textbf{where}\ (l,\ \ \mathit{ls}) \quad = \mathit{parse}\ cs' \\
\qquad\qquad\quad (r,\ \ \mathit{rs}) \quad = \mathit{parse}\ \mathit{ls} \\
\qquad\qquad\quad (t_2,\ cs_2) \ = \mathit{parse}\ \mathit{rs} \\
\qquad\qquad\qquad\qquad\vdots \\
\qquad\qquad\quad (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
= \{\mathit{approx}\} \\
\quad \texttt{'B'} : \mathit{approx}\ n\ (\mathit{pretty}'\ l +\!\!+ \mathit{pretty}'\ r +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ cs_m) \\
\quad\quad \textbf{where}\ (l,\ \ \mathit{ls}) \quad = \mathit{parse}\ cs' \\
\qquad\qquad\quad (r,\ \ \mathit{rs}) \quad = \mathit{parse}\ \mathit{ls} \\
\qquad\qquad\quad (t_2,\ cs_2) \ = \mathit{parse}\ \mathit{rs} \\
\qquad\qquad\qquad\qquad\vdots \\
\qquad\qquad\quad (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
= \{\mathit{pp}_{m+1}\} \\
\quad \texttt{'B'} : \mathit{approx}\ n\ (\mathit{pp}_{m+1}\ cs') \\
\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
\quad \texttt{'B'} : \mathit{approx}\ n\ (\mathit{id}\ cs') \\
= \{\mathit{id}, \mathit{approx}\} \\
\quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\texttt{'B'} : cs')
\end{array}
$$

Hence we have yet again proved (2), this time using the approximation lemma.

Parse after Pretty. Let us now turn to (1'). We want to verify that $\mathit{parse} \circ \mathit{pretty} = \mathit{strictify} :: (T, \mathit{String}) \to (T, \mathit{String})$ holds. This can be done using the approximation lemma as given in equivalence (3). To ease the presentation we will use the following helper function:

$$
\begin{array}{l}
\mathit{approxP} :: \mathit{Nat} \to (T, a) \to (T, a) \\
\mathit{approxP}\ n\ (t, a) = (\mathit{approx}\ n\ t, a)
\end{array}
$$

Using this function we can formulate the approximation lemma as

$$
\mathit{p1} = \mathit{p2} \quad \text{iff} \quad \forall n \in \mathit{Nat}_{\text{fin}}\,.\ \ \mathit{approxP}\ n\ \mathit{p1} = \mathit{approxP}\ n\ \mathit{p2} \tag{13}
$$

for arbitrary pairs $\mathit{p1}, \mathit{p2} :: (T, \psi)$, where ψ is an arbitrary type. In our case $\psi = \mathit{String}$, $\mathit{p1} = (\mathit{parse} \circ \mathit{pretty})\ p$ and $\mathit{p2} = \mathit{strictify}\ p$ for an arbitrary pair $p :: (T, \mathit{String})$.

The proof proceeds by induction over n as usual; and as usual we first have to observe that $\mathit{parse} \circ \mathit{pretty}$ and $\mathit{strictify}$ are both distinct from $\bot$. The case $n = \mathit{Zero}$ is trivial. Now assume that we have proved $\mathit{approxP}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ p) = \mathit{approxP}\ n\ (\mathit{strictify}\ p)$ for some $n \in \mathit{Nat}_{\text{fin}}$ and all $p :: (T, \mathit{String})$. (All p since we can change the order of the universal quantifiers like we did to arrive at inequality (12).) We prove the corresponding statement for $\mathit{Succ}\ n$ by case analysis on p. All cases except for the one where $p = (B\ l\ r, \mathit{cs})$ for arbitrary subtrees $l, r :: T$ and an arbitrary $\mathit{cs} :: \mathit{String}$ are straightforward and omitted, so we go directly to the last case:

$$
\begin{array}{ll}
 & \mathit{approxP}\ (\mathit{Succ}\ n)\ ((\mathit{parse} \circ \mathit{pretty})\ (B\ l\ r, \mathit{cs})) \\
= & \{\circ,\ \mathit{pretty},\ \mathit{pretty}',\ +\!\!+,\ +\!\!+ \text{ associative}\} \\
 & \mathit{approxP}\ (\mathit{Succ}\ n)\ (\mathit{parse}\ (\texttt{'B'} : \mathit{pretty}'\ l +\!\!+ \mathit{pretty}'\ r +\!\!+ \mathit{cs})) \\
= & \{\mathit{parse},\ \mathit{pretty},\ \circ\} \\
 & \mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where}\ (l',\ \mathit{cs}')\ = (\mathit{parse} \circ \mathit{pretty})\ (l, \mathit{pretty}'\ r +\!\!+ \mathit{cs}) \\
 & \qquad\qquad\ (r',\ \mathit{cs}'') = \mathit{parse}\ \mathit{cs}' \\
= & \{\mathit{approxP},\ \mathit{approx}\} \\
 & (B\ (\mathit{approx}\ n\ l')\ (\mathit{approx}\ n\ r'), \mathit{cs}'') \\
 & \quad \textbf{where}\ (l',\ \mathit{cs}')\ = (\mathit{parse} \circ \mathit{pretty})\ (l, \mathit{pretty}'\ r +\!\!+ \mathit{cs}) \\
 & \qquad\qquad\ (r',\ \mathit{cs}'') = \mathit{parse}\ \mathit{cs}' \\
= & \{\text{Push } \mathit{approx}\ n \text{ through the pairs, turning it into } \mathit{approxP}\ n\} \\
 & (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where}\ (l',\ \mathit{cs}')\ = \mathit{approxP}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ (l, \mathit{pretty}'\ r +\!\!+ \mathit{cs})) \\
 & \qquad\qquad\ (r',\ \mathit{cs}'') = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}') \\
= & \{\text{Inductive hypothesis}\} \\
 & (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where}\ (l',\ \mathit{cs}')\ = \mathit{approxP}\ n\ (\mathit{strictify}\ (l, \mathit{pretty}'\ r +\!\!+ \mathit{cs})) \\
 & \qquad\qquad\ (r',\ \mathit{cs}'') = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}') \\
= & \{\mathit{strictify}\} \\
 & (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where}\ (l',\ \mathit{cs}')\ = \quad \mathit{approxP}\ n\ (l\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+ \mathit{cs})) \\
 & \qquad\qquad\ (t',\ \mathit{tTotal}) = \mathit{strictify}'\ l \\
 & \qquad\qquad\ (r',\ \mathit{cs}'') = \quad \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}')
\end{array}
$$

The proof proceeds by case analysis on l. We omit the cases $l = \bot$ and $l = L$ and go to the last case, $l = B\ l_1\ r_1$ for arbitrary subtrees $l_1, r_1 :: T$:

$$
\begin{array}{ll}
 & (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where} \\
 & \quad (l',\ \mathit{cs}') \qquad = \mathit{approxP}\ n\ (B\ l_1\ r_1\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+ \mathit{cs})) \\
 & \quad (t',\ \mathit{tTotal}) = \mathit{strictify}'\ (B\ l_1\ r_1) \\
 & \quad (r',\ \mathit{cs}'') \qquad = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}') \\
= & \{\mathit{seq},\ \mathit{strictify}',\ \textbf{where}\} \\
 & (B\ l'\ r', \mathit{cs}'') \\
 & \quad \textbf{where} \\
 & \quad (l',\ \mathit{cs}') \qquad = \mathit{approxP}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+ \mathit{cs}) \\
 & \quad (l_1',\ \mathit{lTotal})\ = \mathit{strictify}'\ l_1 \\
 & \quad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
 & \quad (r',\ \mathit{cs}'') \qquad = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}') \\
= & \{\mathit{approxP},\ \textbf{where}\} \\
 & (B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', \mathit{cs}'') \\
 & \quad \textbf{where} \\
 & \quad (l_1',\ \mathit{lTotal})\ = \mathit{strictify}'\ l_1 \\
 & \quad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
 & \quad (r',\ \mathit{cs}'') \qquad = \mathit{approxP}\ n\ (\mathit{parse}\ ((\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+ \mathit{cs}))
\end{array}
$$

Now we have two cases, depending on whether $lTotal$ ‘seq‘ $rTotal$, i.e. snd $(strictify'\ l_1)$ ‘seq‘ snd $(strictify'\ r_1)$, equals $\bot$ or not. We omit the case where the equality holds and concentrate on the case where $lTotal$‘seq‘$rTotal = () \neq \bot$:

$$
\begin{array}{l}
\quad (B\ (approx\ n\ (B\ l_1'\ (lTotal\ \text{‘}seq\text{‘}\ r_1')))\ r', cs'') \\
\qquad \textbf{where}\ (l_1', lTotal) = strictify'\ l_1 \\
\qquad\qquad\quad\ (r_1', rTotal) = strictify'\ r_1 \\
\qquad\qquad\quad\ (r', cs'') \quad\ = approxP\ n\ (parse\ (()\ \text{‘}seq\text{‘}\ pretty'\ r +\!\!+\ cs)) \\
= \{seq, pretty, \circ\} \\
\quad (B\ (approx\ n\ (B\ l_1'\ (lTotal\ \text{‘}seq\text{‘}\ r_1')))\ r', cs'') \\
\qquad \textbf{where}\ (l_1', lTotal) = strictify'\ l_1 \\
\qquad\qquad\quad\ (r_1', rTotal) = strictify'\ r_1 \\
\qquad\qquad\quad\ (r', cs'') \quad\ = approxP\ n\ ((parse \circ pretty)\ (r, cs)) \\
= \{\text{Inductive hypothesis}\} \\
\quad (B\ (approx\ n\ (B\ l_1'\ (lTotal\ \text{‘}seq\text{‘}\ r_1')))\ r', cs'') \\
\qquad \textbf{where}\ (l_1', lTotal) = strictify'\ l_1 \\
\qquad\qquad\quad\ (r_1', rTotal) = strictify'\ r_1 \\
\qquad\qquad\quad\ (r', cs'') \quad\ = approxP\ n\ (strictify\ (r, cs)) \\
= \{\text{Push } approxP\ n \text{ through the pair, turning it into } approx\ n\} \\
\quad (B\ (approx\ n\ (B\ l_1'\ (lTotal\ \text{‘}seq\text{‘}\ r_1')))\ (approx\ n\ r'), cs'') \\
\qquad \textbf{where}\ (l_1', lTotal) = strictify'\ l_1 \\
\qquad\qquad\quad\ (r_1', rTotal) = strictify'\ r_1 \\
\qquad\qquad\quad\ (r', cs'') \quad\ = strictify\ \ (r, cs) \\
= \{approx, approxP\} \\
\quad approxP\ (Succ\ n)\ (B\ (B\ l_1'\ (lTotal\ \text{‘}seq\text{‘}\ r_1'))\ r', cs'') \\
\qquad \textbf{where}\ (l_1', lTotal) = strictify'\ l_1 \\
\qquad\qquad\quad\ (r_1', rTotal) = strictify'\ r_1 \\
\qquad\qquad\quad\ (r', cs'') \quad\ = strictify\ \ (r, cs)
\end{array}
$$

The rest of the proof consists of transforming the expression above to $approxP$ $(Succ\ \ n)$ $(strictify\ \ (B\ \ (B\ \ l_1\ \ r_1)\ \ r, cs))$. This is relatively straightforward and omitted. Thus we have, yet again, proved (1’).

9 Discussion and Future Work

In this paper we have investigated how different verification methods can handle partial and infinite values in a simple case study about data conversion. We have used random testing, fixpoint induction and the approximation lemma.

Using $isBottom$ and $approx$ for testing in the presence of partial and infinite values is not fool proof but works well in practice. The approach is not that original; testing using $isBottom$ and $take$ is (indirectly) mentioned already in the original QuickCheck paper [2]. However, testing using $approx$ has probably not been done before. Furthermore, the functionality of $\hat{=}$ and $\hat{\sqsubseteq}$ has not been provided by any (widespread) library.

The two methods used for proving the properties (1’) and (2) have different qualities. Fixpoint induction required us to rewrite both the functions and the properties. Furthermore one property did not hold for the base case, so it had to be rewritten (7'), and proving the base case required some tedious but straightforward work. On the other hand, once the initial work had been completed the "actual proofs" were comparatively short. The corresponding "actual proofs" were longer when using the approximation lemma. The reason for this is probably that the approximation lemma requires that the function *approx* is "pushed" inside the expressions to make it possible to apply the inductive hypothesis. For fixpoint induction that is not necessary. For instance, when proving (1') using the approximation lemma we had to go one level further down in the tree when performing case analysis, than in the corresponding proof using fixpoint induction. This was in order to be able to use the inductive hypothesis.

Nevertheless, the "actual proofs" are not really what is important. They mostly consist of performing a case analysis, evaluating both sides of the (in-)equality being proved as far as possible and then, if the proof is not finished yet, choosing a new expression to perform case analysis on. The most important part is really finding the right inductive hypothesis. (Choosing the right expression for case analysis is also important, but easier.) Finding the right inductive hypothesis was easier when using fixpoint induction than when using the approximation lemma. Take the proofs of (2), for instance. When using fixpoint induction almost no thought was needed to come up with the inductive hypothesis, whereas when using the approximation lemma we had to come up with the complex hypothesis based on property (10), the one involving pp_m. The reason was the same as above; *approx* has to be in the right position. It is of course possible that easier proofs exist.

It is also possible that there are other proof methods which work better than the ones used here. Coinduction and fusion, two other methods mentioned in Gibbons' and Hutton's tutorial [4], might belong to that category. We have made some attempts at using unfold fusion. Due to the nature of the programs the standard fusion method seems inapplicable, though; a monadic variant is a better fit. The programs can be transformed into monadic variants (which of course carries extra proof obligations). We have not yet figured out where to go from there, though. For instance, the monadic anamorphism fusion law [10, Equation (17)] only applies to a restrictive class of monads, and our "monad" does not even satisfy all the monad laws (compare Sect. 3).

Above we have compared different proof techniques in the case where we allow infinite and partial input. Let us now reflect on whether one should consider anything but finite, total values. The proofs of (1') and (2) valid for all inputs were considerably longer than the ones for (1) and (2) limited to finite (and in one case total) input, especially if one takes into account all work involved in rewriting the properties and programs. It is not hard to see why people often ignore partial and infinite input; handling it does seem to require nontrivial amounts of extra work.

However, as argued in Sect. 1 we often need to reason about infinite values. Furthermore, in reality, bottoms do occur; *error* is used, cases are left out from case expressions, and sometimes functions do not reach a weak head normal form even if they are applied to total input (for instance we have *reverse* $[1\,..] = \bot$).

Another reason for including partial values is that in our setting of equational reasoning it is easier to use a known identity if the identity is valid without a precondition stating that the input has to be total. Of course, proving the identity without this precondition is only meaningful if the extra work involved is less than the accumulated work needed to verify the precondition each time the identity is used. This extra work may not amount to very much, though. Even if we were to ignore bottoms, we would still sometimes need to handle infinite values, so we would have to use methods like those used in this text. In this case the marginal cost for also including bottoms would be small.

Another approach is to settle for approximate results by e.g. assuming that $\lambda x \rightarrow \bot$ is $\bot$ when reasoning about programs. These results would be practically useful; we might get some overly conservative results if we happened to evaluate $seq\ (\lambda x \rightarrow \bot)$, but nothing worse would happen. On the other hand, many of the caveats mentioned in Sect. 3 would vanish. Furthermore most people tend to ignore these issues when doing ordinary programming, so in a sense an approximate semantics is already in use. The details of an approximate semantics for Haskell still need to be worked out, though. We believe that an approach like this will make it easier to scale up the methods used in this text to larger programs.

Acknowledgements

We would like to thank Andreas Abel, Ulf Norell and the anonymous referees for some helpful comments and/or discussions.

References

[1] Manuel Chakravarty et al. *The Haskell 98 Foreign Function Interface 1.0, An Addendum to the Haskell 98 Report*, 2003. Available online at `http://www.haskell.org/definition/`.

[2] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

[3] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.

[4] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. Submitted to Fundamenta Informaticae Special Issue on Program Transformation. Available online at `http://www.cs.nott.ac.uk/~gmh/bib.html`, March 2004.

[5] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[6] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, August 2001.

[7] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[8] Patricia Johann and Janis Voigtländer. Free theorems in the presence of *seq*. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 99–110. ACM Press, 2004.

[9] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.

[10] Alberto Pardo. Monadic corecursion – definition, fusion laws, and applications –. In Bart Jacobs, Larry Moss, Horst Reichel, and Jan Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 11. Elsevier, 2000.

[11] Simon Peyton Jones. *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series: Computer & Systems Sciences*, chapter Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, pages 47–96. IOS Press, 2001. Updated version available online at `http://research.microsoft.com/~simonpj/`.

[12] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[13] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):175–234, March 1996.

[14] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. W.C. Brown, Dubuque, Iowa, 1988.

A QuickCheck Generators

The QuickCheck generators used in this text are defined as follows:

```
tree :: Gen T
tree = frequency [(6, liftM2 B tree tree),
                  (2, return L),
                  (1, return ⊥)]

string :: Gen String
string = frequency [(1, bottomString),
                    (1, finiteString),
                    (1, infiniteString),
                    (3, treeString)]
  where
  bottomString   = liftM2 approx        arbitrary infiniteString
  finiteString   = liftM2 (take ∘ abs)  arbitrary infiniteString
  infiniteString = liftM2 (:)           char      infiniteString
  treeString     = tree ≫= return ∘ pretty′

char :: Gen Char
char = frequency [(10, return 'B'),
                  (10, return 'L'),
                  (1,  return '?'),
                  (1,  return ⊥)]

pair :: Gen (T, String)
pair = frequency [(50, liftM2 (,) tree string),
                  (1,  return ⊥)]
```

A straightforward *Arbitrary* instance for *Nat* (yielding only total, finite values) is also required.

The generator *tree* is defined so that the generated trees have a probability of $\frac{1}{2}$ of being finite, and the finite trees have an expected depth of 2.[2] We do not generate any total, infinite trees. The reason is that some of the tests above do not terminate for such trees, as shown in Sect. 5.

To get a good mix of finite and infinite partial strings the *string* generator is split up into four cases. The last case ensures that some strings that actually represent trees are also included. It would not be a problem to include total, infinite strings, but we do not want to complicate the definitions above too much, so they are also omitted.

Finally the *pair* generator constructs pairs using *tree* and *string*, forcing some pairs to be $\perp$.

By using *collect* we have observed that the actual distributions of generated values correspond to our expectations.

[2] Assuming that QuickCheck uses a random number generator that yields independent values from a uniform distribution.