Thesis for the degree of Licentiate of Engineering

Precise Reasoning About Non-strict Functional Programs

How to Chase Bottoms, and How to Ignore Them

Nils Anders Danielsson

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2005

Precise Reasoning About Non-strict Functional Programs
How to Chase Bottoms, and How to Ignore Them
Nils Anders Danielsson

Technical report 7L
ISSN 1652-876X
Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers, Göteborg, Sweden, 2005

Precise Reasoning About Non-strict Functional Programs

How to Chase Bottoms, and How to Ignore Them

Nils Anders Danielsson

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

Abstract

This thesis consists of two parts. Both concern reasoning about non-strict functional programming languages with partial and infinite values and lifted types, including lifted function spaces.

The first part is a case study in program verification: We have written a simple parser and a corresponding pretty-printer in 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 makes this exercise interesting. We have tackled the problem in different ways, and report on the merits of those approaches. More specifically, first a method for testing properties of programs in the presence of partial and infinite values is described. By testing before proving we avoid wasting time trying to prove statements that are not valid. Then it is proved that the programs we have written are in fact (more or less) inverses using first fixpoint induction and then the approximation lemma.

Using the proof methods described in the first part can be cumbersome. As an alternative, the second part justifies reasoning about non-total (partial) functional languages using methods seemingly only valid for total ones. Two languages are defined, one total and one partial, with identical syntax. A partial equivalence relation is then defined, the domain of which is the total subset of the partial language. It is proved that if two closed terms have the same semantics in the total language, then they have related semantics in the partial language.

Keywords: Functional programming, equational reasoning, non-strict, partial, infinite, lifted, inductive, coinductive.

Acknowledgements

I would like to thank Patrik for being an enthusiastic supervisor.

I would also like to thank my coauthors and others who have contributed to this work, either directly or indirectly, and my fellow PhD students who have provided distractions from this work.

Finally I would like to thank Maria for being very supportive.

Introduction

Most people that use computers have come across programs that fail to function correctly; they crash, or do something else which they are not supposed to do.

One way of avoiding such problems is to formally verify the correctness of a program before it is used. First one has to decide what, exactly, it is that the program should do, and then, by inspecting the source code in some way, one should come up with a mathematically convincing argument for why the program behaves as it should.

In general this is not an easy task, neither the specification nor the proof of correctness. Some programming languages are claimed to make the proofs easier to perform, though. One example is functional programming languages, where programs consist of functions that are similar to ordinary mathematical functions.

An interesting aspect of some functional languages is that they have natural support for manipulating infinite values. Furthermore they typically handle values that are only partially defined. Unfortunately, proofs about functional programs often ignore details related to partial and infinite values. That is addressed by the first paper contained in this thesis:

Page 9 *Chasing Bottoms, A Case Study in Program Verification in the Presence of Partial and Infinite Values*, written together with Patrik Jansson. This is an extended version of the paper published in the *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004, LNCS 3125, Springer-Verlag, 2004.*

The value that is totally undefined is called bottom, hence the title. The paper is a case study where we explore how one can go about testing and proving even in the presence of partial and infinite values. More concretely, simple programs for writing and reading so-called binary trees are presented, and it is shown using several methods what happens when reading something previously written, and vice versa.

One of the conclusions of the first paper is that explicitly handling infinite and partial values requires a lot of work. Infinite values are sometimes useful, but that is seldom true for the partially defined ones, hence having to deal with them is cumbersome. That is addressed by the second paper:

Page 49 *Fast and Loose Reasoning is Morally Correct*, written together with Jeremy Gibbons and John Hughes. Not yet submitted for publication.

The paper explores whether one can reason in a fast and loose way—ignoring the bottoms, pretending that values are totally defined (and perhaps finite)—but still get useful results. Theory is presented that shows how this can be done. Some examples using the theory are also included, showing cases both in which the theory leads to easier proofs, and in which other methods are to be preferred.

Many proofs used for the theory in the second paper are only sketched. Full proofs can be found at

`http://www.cs.chalmers.se/~nad/`.

A library which can be used to chase bottoms in Haskell programs can also be found there. (Haskell is a functional programming language figuring prominently in both papers.)

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 [Hug89]. 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.

[*]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."

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 [GH05].

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 [JJ02]. The case study in this paper uses cut down versions of those programs (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 [PJ03], 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:

$$\begin{aligned}
&\mathit{pretty} :: (T, \mathit{String}) \to \mathit{String}\\
&\mathit{pretty}\ (t, \mathit{cs}) = \mathit{pretty}'\ t +\!\!+\ \mathit{cs}
\end{aligned}$$

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 [PJ03], 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 standard formal semantics for the language to verify it against. However, we think that the existing knowledge of Haskell semantics [HSH02, HK, Fax02] can be combined into a consistent framework for equational reasoning.

Thus 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 [San96] based on an operational model, we still believe that the proof steps would be essentially the same.

Now on to the caveats (see also [HSH02]). 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 $\textbf{let}\ (a, b) = x\ \textbf{in}\ g\ (a, b)$ 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 $\textbf{let}\ (a, b) = \bot\ \textbf{in}\ (a, b)$ 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 [PJ03]:

$$\begin{aligned} &\mathit{seq} :: a \to b \to b \\ &\mathit{seq}\ \bot\ b = \bot \\ &\mathit{seq}\ a\ \ b = b \end{aligned}$$

(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 [JV04], but that does not directly affect us because our functions are not polymorphic.

Another caveat, also related to *seq*, is that $f = \lambda \mathit{True}\ x \rightarrow x$ is not identical to $f' = \lambda \mathit{True} \rightarrow \lambda x \rightarrow x$. By careful inspection of Haskell's pattern matching semantics [PJ03] we can see that $f\ \mathit{False} = \lambda x \rightarrow \bot$ while $f'\ \mathit{False} = \bot$, since the function f is interpreted as

$$\begin{array}{l} \lambda a \rightarrow \lambda b \rightarrow \textbf{case}\ (a, b)\ \textbf{of} \\ \quad (\mathit{True}, x) \rightarrow x \end{array}$$

whereas the function f' is interpreted as

$$\begin{array}{l} \lambda a \rightarrow \textbf{case}\ a\ \textbf{of} \\ \quad \mathit{True} \rightarrow \lambda x \rightarrow x\ . \end{array}$$

This also applies if f and f' are defined by $f\ \mathit{True}\ x = x$ and $f'\ \mathit{True} = \lambda x \rightarrow x$. We do not get any problems if the first pattern is a simple variable, though. In this paper 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.

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

and

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

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 $\perp$, and then we continue using structural induction. The inductive hypothesis used is

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

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 $cs :: \mathit{String}$):

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

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**.)

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

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

Hence we have proved using structural induction that $(\mathit{parse} \circ \mathit{pretty})\ (t, cs) = (t, cs)$ for all finite, total $t :: T$ and for all $cs :: \mathit{String}$. 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. The proof is by a strong form of induction. When proving the property for a string cs we can assume that it holds for all tails cs' of cs, and also all strings $cs'' \sqsubseteq cs'$. The first case is for a $cs :: \mathit{String}$ satisfying $\mathit{head}\ cs \notin \{\texttt{'L'}, \texttt{'B'}\}$:

$$
\begin{aligned}
&\quad (\mathit{pretty} \circ \mathit{parse})\ cs\\
&= \{\circ\}\\
&\quad \mathit{pretty}\ (\mathit{parse}\ cs)
\end{aligned}
$$

$= \{\textit{parse},\ \textit{head cs} \notin \{\texttt{'L'}, \texttt{'B'}\}\}$

$\quad \textit{pretty}\ \bot$

$= \{\textit{pretty}\}$

$\quad \bot :: \textit{String}$

$\sqsubseteq \{\bot \text{ is the least element}\}$

$\quad \textit{cs}$

Second case, $\textit{cs} :: \textit{String}$, $\textit{head cs} = \texttt{'L'}$, i.e. $\textit{cs} = \texttt{'L'} : \textit{cs}_1$ for some $\textit{cs}_1 :: \textit{String}$:

$\quad (\textit{pretty} \circ \textit{parse})\ (\texttt{'L'} : \textit{cs}_1)$

$= \{\circ,\ \textit{parse}\}$

$\quad \textit{pretty}\ (L, \textit{cs}_1)$

$= \{\textit{pretty}\}$

$\quad \textit{pretty}'\ L +\!\!+\ \textit{cs}_1$

$= \{\textit{pretty}',\ +\!\!+\}$

$\quad \texttt{'L'} : \textit{cs}_1$

Last case, $\textit{cs} :: \textit{String}$, $\textit{head cs} = \texttt{'B'}$, i.e. $\textit{cs} = \texttt{'B'} : \textit{cs}_1$ for some (partial and finite) $\textit{cs}_1 :: \textit{String}$:

$\quad (\textit{pretty} \circ \textit{parse})\ (\texttt{'B'} : \textit{cs}_1)$

$= \{\circ,\ \textit{parse}\}$

$\quad \textit{pretty}\ (B\ l\ r, \textit{cs}_1'')$
$\qquad \textbf{where}\ (l,\ \textit{cs}_1') = \textit{parse}\ \textit{cs}_1$
$\qquad\qquad\quad (r, \textit{cs}_1'') = \textit{parse}\ \textit{cs}_1'$

$= \{\textit{pretty},\ \textit{pretty}',\ +\!\!+ \text{ associative}\}$

$\quad \texttt{"B"} +\!\!+\ \textit{pretty}'\ l +\!\!+\ \textit{pretty}'\ r +\!\!+\ \textit{cs}_1''$
$\qquad \textbf{where}\ (l,\ \textit{cs}_1') = \textit{parse}\ \textit{cs}_1$
$\qquad\qquad\quad (r, \textit{cs}_1'') = \textit{parse}\ \textit{cs}_1'$

$= \{\textit{pretty}\}$

$\quad \texttt{"B"} +\!\!+\ \textit{pretty}'\ l +\!\!+\ \textit{pretty}\ (r, \textit{cs}_1'')$
$\qquad \textbf{where}\ (l,\ \textit{cs}_1') = \textit{parse}\ \textit{cs}_1$
$\qquad\qquad\quad (r, \textit{cs}_1'') = \textit{parse}\ \textit{cs}_1'$

$= \{\textbf{where},\ \textit{pretty}\ \bot = \textit{pretty}\ (\bot, \bot),\ \circ\}$

$\quad \texttt{"B"} +\!\!+\ \textit{pretty}'\ l +\!\!+\ (\textit{pretty} \circ \textit{parse})\ \textit{cs}_1'$
$\qquad \textbf{where}\ (l, \textit{cs}_1') = \textit{parse}\ \textit{cs}_1$

$\sqsubseteq$ {Inductive hypothesis (see below), monotonicity}

$\texttt{"B"} +\!\!+\ \mathit{pretty}'\ l +\!\!+\ \mathit{id}\ \mathit{cs}_1'$
$\quad \textbf{where}\ (l, \mathit{cs}_1') = \mathit{parse}\ \mathit{cs}_1$

$= \{\mathit{id}, \mathit{pretty}, \textbf{where}, \mathit{pretty}\ \bot = \mathit{pretty}\ (\bot, \bot), \circ\}$

$\texttt{"B"} +\!\!+\ (\mathit{pretty} \circ \mathit{parse})\ \mathit{cs}_1$

$\sqsubseteq$ {Inductive hypothesis, monotonicity}

$\texttt{"B"} +\!\!+\ \mathit{id}\ \mathit{cs}_1$

$= \{\mathit{id}, +\!\!+\}$

$\texttt{'B'} : \mathit{cs}_1$

A lemma is required which shows that we can apply the inductive hypothesis for $\mathit{snd}\ (\mathit{parse}\ \mathit{cs}_1)$ whenever we can do it for cs_1. We omit the proof of the lemma, which is by strong induction over cs_1.

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:

$(\mathit{parse} \circ \mathit{pretty})\ (\bot, \mathit{cs})$

$= \{\circ, \mathit{pretty}\}$

$\mathit{parse}\ (\mathit{pretty}'\ \bot +\!\!+\ \mathit{cs})$

$= \{\mathit{pretty}', +\!\!+\}$

$\mathit{parse}\ \bot$

$= \{\mathit{parse}\}$

$\bot :: (T, \mathit{String})$

$\neq$ {(,) is not strict}

$(\bot, \mathit{cs}) :: (T, \mathit{String})$

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 [HG01]. 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 $t_1, t_2 :: T$,

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

Here $\mathit{Nat}_{\text{fin}}$ stands for the total and finite values of type *Nat*, i.e. $\mathit{Nat}_{\text{fin}}$ 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 $\mathit{take} :: \mathit{Int} \to [a] \to [a]$ 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 [HG01]. 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 finite but partial values. Thus even if we were dealing with total values only, we would still need to include $\bot$ in our tests. Generating the value $\bot$ is easily accomplished:

$$\begin{array}{l} \bot :: a \\ \bot = \mathit{error}\ \texttt{"\_|\_"} \end{array}$$

(Note that the same notation is used for the expression that generates a $\bot$ 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 [PJ01]. Furthermore it would be nice if we could perform these tests in pure code, such as QuickCheck [CH00] properties (see below). This can only be accomplished by using the decidedly unsafe function *unsafePerformIO* $:: IO\ \ a \to a$ [C$^+$03, PJ01]. The resulting function *isBottom* $:: a \to Bool$[1] has to be used with care; it only detects a $\bot$ 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
```

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 [LPJ03].

[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.

QuickCheck is a library for defining and testing properties of Haskell functions [CH00]. By using the framework developed above we can now give QuickCheck implementations of properties (1) and (2):

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

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:

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

(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 second component of the output pair on the left-hand side is a $\bot$ of the non-terminating kind which we cannot detect. 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''

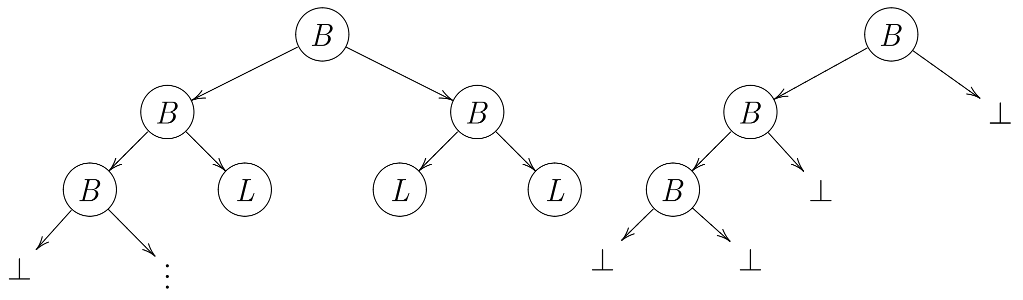

Figure 1: With the left tree called t, the right tree is $t' = (\mathit{fst} \circ \mathit{parse} \circ \mathit{pretty}')\ t$.

will not be generated until that subexpression has produced any output in its second 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 $\mathit{propFigure}$ 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 $\mathit{strictify}$ for id that makes

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

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

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

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:

$$\begin{aligned}&\mathit{strictify}' :: T \to (T, ())\\ &\mathit{strictify}'\ L \qquad\ = (L, ())\\ &\mathit{strictify}'\ (B\ l\ r) = (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\\ &\quad \textbf{where}\ (l', \mathit{lTotal}) = \mathit{strictify}'\ l\\ &\qquad\qquad (r', \mathit{rTotal}) = \mathit{strictify}'\ r\end{aligned}$$

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

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

$$\begin{aligned}&\mathit{prop}_1'\ n = \mathit{forAll}\ \mathit{pair}\ (\lambda p \to\\ &\quad \mathit{approxPair}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ p) \mathrel{\hat{=}} \mathit{approxPair}\ n\ (\mathit{strictify}\ p))\end{aligned}$$

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 [GH05, Sch88] 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:

$$\begin{aligned}&\mathit{fix} :: (a \to a) \to a\\&\mathit{fix}\ f = f\ (\mathit{fix}\ f)\end{aligned}$$

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*:

$$\begin{aligned}&\mathit{pretty}'_{\mathit{fix}} :: T \to \mathit{String}\\&\mathit{pretty}'_{\mathit{fix}} = \mathit{fix}\ \mathit{pretty}_{\mathit{step}}\\[4pt]&\mathit{pretty}_{\mathit{step}} :: (T \to \mathit{String}) \to T \to \mathit{String}\\&\mathit{pretty}_{\mathit{step}}\ p\ L \qquad\ \ = \texttt{"L"}\\&\mathit{pretty}_{\mathit{step}}\ p\ (B\ l\ r) = \texttt{"B"} +\!\!+\ p\ l +\!\!+\ p\ r\end{aligned}$$

$$\begin{aligned}&\mathit{parse}_{\mathit{fix}} :: \mathit{String} \to (T, \mathit{String})\\&\mathit{parse}_{\mathit{fix}} = \mathit{fix}\ \mathit{parse}_{\mathit{step}}\\[4pt]&\mathit{parse}_{\mathit{step}} :: (\mathit{String} \to (T, \mathit{String})) \to \mathit{String} \to (T, \mathit{String})\\&\mathit{parse}_{\mathit{step}}\ p'\ (\texttt{'L'} : cs) = (L, cs)\\&\mathit{parse}_{\mathit{step}}\ p'\ (\texttt{'B'} : cs) = (B\ l\ r, cs'')\\&\quad \textbf{where}\ (l,\ cs') \ = p'\ cs\\&\qquad\qquad\ \ (r,\ cs'') = p'\ cs'\end{aligned}$$

$$
\begin{aligned}
&\mathit{strictify}'_{\mathit{fix}} :: T \to (T, ())\\
&\mathit{strictify}'_{\mathit{fix}} = \mathit{fix}\ \mathit{strictify}_{\mathit{step}}\\
&\mathit{strictify}_{\mathit{step}} :: (T \to (T, ())) \to T \to (T, ())\\
&\mathit{strictify}_{\mathit{step}}\ s\ L \qquad\quad = (L, ())\\
&\mathit{strictify}_{\mathit{step}}\ s\ (B\ l\ r) = (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\\
&\quad \mathbf{where}\ (l', \mathit{lTotal}) = s\ l\\
&\qquad\qquad\ (r', \mathit{rTotal}) = s\ r
\end{aligned}
$$

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

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

and

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

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 $t \neq \bot$:

$$
\begin{aligned}
&\quad (\bot \circ \mathit{pretty}_{\mathit{fix}}\ \bot)\ (t, \mathit{cs})\\
&= \{\circ, \bot\}\\
&\quad \bot :: (T, \mathit{String})
\end{aligned}
$$

$\neq \{seq,\ t \neq \bot,\ (,) \text{ is not strict}\}$

$\quad t\ \text{`}seq\text{`}\ (\bot, \bot) :: (T, String)$

$= \{seq\}$

$\quad t\ \text{`}seq\text{`}\ (\bot, \bot\ \text{`}seq\text{`}\ cs) :: (T, String)$

$= \{\textbf{where}, \text{pattern matching}\}$

$\quad t\ \text{`}seq\text{`}\ (t', tTotal\ \text{`}seq\text{`}\ cs) :: (T, String)$
$\quad\quad \textbf{where}\ (t', tTotal) = \bot$

$= \{\bot\}$

$\quad t\ \text{`}seq\text{`}\ (t', tTotal\ \text{`}seq\text{`}\ cs) :: (T, String)$
$\quad\quad \textbf{where}\ (t', tTotal) = \bot\ t$

$= \{strictify_{fix}\}$

$\quad strictify_{fix}\ \bot\ (t, cs)$

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

$$P_1'(p, p', s) = P_1(pretty_{step}\ p, parse_{step}\ p', strictify_{step}\ s), \tag{7'}$$

i.e. $P_1'(p, p', s)$ is equivalent to

$$parse_{step}\ p' \circ pretty_{fix}\ (pretty_{step}\ p) = strictify_{fix}\ (strictify_{step}\ s). \tag{9}$$

With P_1' we avoid the troublesome base case since $P_1'(\bot, \bot, \bot)$ is equivalent to $P_1(pretty_{step}\ \bot, parse_{step}\ \bot, strictify_{step}\ \bot)$.

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 [Sch88]. As we have already stated above, verifying formally that $P_1'(fix\ pretty_{step}, fix\ parse_{step}, fix\ strictify_{step})$ is equivalent to (1') and similarly that $P_2(fix\ pretty_{step}, fix\ parse_{step})$ is equivalent to (2) requires more work and is omitted.

Pretty after Parse. Having formulated the predicates and verified the base cases we now move on to the main work; the step cases. 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 $pretty_{step}$ and $parse_{step}$ are strictness-preserving. Assume now that $P_2(p, p')$ is valid for strict $p :: T \to String$ and $p' :: String \to (T, String)$. We have to show that $P_2(pretty_{step}\ p, parse_{step}\ p')$ 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 $cs_1 :: \mathit{String}$:

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

$= \{$**where**, p strict implies that $\mathit{pretty}_{\mathit{fix}}\ p\ \bot = \mathit{pretty}_{\mathit{fix}}\ p\ (\bot, \bot)$, $\circ\}$

$\quad$ `"B"` $+\!\!+\ p\ l +\!\!+ (\mathit{pretty}_{\mathit{fix}}\ p \circ p')\ \mathit{cs}_1'$

$\quad\quad$ **where** $(l, \mathit{cs}_1') = p'\ \mathit{cs}_1$

$\sqsubseteq$ {Inductive hypothesis, monotonicity}

$\quad$ `"B"` $+\!\!+\ p\ l +\!\!+ \mathit{id}\ \mathit{cs}_1'$

$\quad\quad$ **where** $(l, \mathit{cs}_1') = p'\ \mathit{cs}_1$

$= \{\mathit{id}, \mathit{pretty}_{\mathit{fix}}$, **where**, p strict, $\circ\}$

$\quad$ `"B"` $+\!\!+ (\mathit{pretty}_{\mathit{fix}}\ p \circ p')\ \mathit{cs}_1$

$\sqsubseteq$ {Inductive hypothesis, monotonicity}

$\quad$ `"B"` $+\!\!+\ \mathit{id}\ \mathit{cs}_1$

$= \{\mathit{id}, +\!\!+, \mathit{id}\}$

$\quad \mathit{id}\ \mathit{cs}$

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. To verify the step case we have to prove

$$P_1'(p_0, p_0', s_0) \Longrightarrow P_1'(\mathit{pretty}_{\mathit{step}}\ p_0, \mathit{parse}_{\mathit{step}}\ p_0', \mathit{strictify}_{\mathit{step}}\ s_0) \qquad (10)$$

for all strict p_0, p_0' and s_0. With $p = \mathit{pretty}_{\mathit{step}}\ p_0$, $p' = \mathit{parse}_{\mathit{step}}\ p_0'$ and $s = \mathit{strictify}_{\mathit{step}}\ s_0$ this simplifies to

$$P_1(p, p', s) \Longrightarrow P_1(\mathit{pretty}_{\mathit{step}}\ p, \mathit{parse}_{\mathit{step}}\ p', \mathit{strictify}_{\mathit{step}}\ s). \qquad (11)$$

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 performed using case analysis, this time on *pair*, an arbitrary element in (T, cs).

First case, $\mathit{pair} = \bot$:

$\quad (\mathit{parse}_{\mathit{step}}\ p' \circ \mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p))\ \bot$

$= \{\circ, \mathit{pretty}_{\mathit{fix}}, \mathit{parse}_{\mathit{step}}\}$

$\quad \bot :: (T, \mathit{String})$

$= \{\mathit{seq}, \mathit{strictify}_{\mathit{fix}}\}$

$\quad \mathit{strictify}_{\mathit{fix}}\ (\mathit{strictify}_{\mathit{step}}\ s)\ \bot :: (T, \mathit{String})$

Second case, $\mathit{pair} = (\bot, \mathit{cs})$ for an arbitrary $\mathit{cs} :: \mathit{String}$:

$$\begin{aligned}
&(\mathit{parse}_{\mathit{step}}\ p' \circ \mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p))\ (\bot, \mathit{cs})\\
=&\ \{\circ,\ \mathit{pretty}_{\mathit{fix}}\}\\
&\mathit{parse}_{\mathit{step}}\ p'\ (\mathit{pretty}_{\mathit{step}}\ p\ \bot \mathbin{+\!\!+} \mathit{cs})\\
=&\ \{\mathit{pretty}_{\mathit{step}},\ \mathbin{+\!\!+},\ \mathit{parse}_{\mathit{step}}\}\\
&\bot :: (T, \mathit{String})\\
=&\ \{\mathit{seq},\ \mathit{strictify}_{\mathit{fix}}\}\\
&\mathit{strictify}_{\mathit{fix}}\ (\mathit{strictify}_{\mathit{step}}\ s)\ (\bot, \mathit{cs})
\end{aligned}$$

Third case, $\mathit{pair} = (L, \mathit{cs})$ for an arbitrary $\mathit{cs} :: \mathit{String}$:

$$\begin{aligned}
&(\mathit{parse}_{\mathit{step}}\ p' \circ \mathit{pretty}_{\mathit{fix}}\ (\mathit{pretty}_{\mathit{step}}\ p))\ (L, \mathit{cs})\\
=&\ \{\circ,\ \mathit{pretty}_{\mathit{fix}}\}\\
&\mathit{parse}_{\mathit{step}}\ p'\ (\mathit{pretty}_{\mathit{step}}\ p\ L \mathbin{+\!\!+} \mathit{cs})\\
=&\ \{\mathit{pretty}_{\mathit{step}},\ \mathbin{+\!\!+},\ \mathit{parse}_{\mathit{step}}\}\\
&(L, \mathit{cs})\\
=&\ \{\mathit{seq},\ \mathbf{where}\}\\
&L\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \mathbf{where}\ (t', \mathit{tTotal}) = (L, ())\\
=&\ \{\mathit{strictify}_{\mathit{step}}\}\\
&L\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \mathbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}_{\mathit{step}}\ s\ L\\
=&\ \{\mathit{strictify}_{\mathit{fix}}\}\\
&\mathit{strictify}_{\mathit{fix}}\ (\mathit{strictify}_{\mathit{step}}\ s)\ (L, \mathit{cs})
\end{aligned}$$

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

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

$= \{\mathit{pretty}_{\mathit{fix}}, \circ\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l', cs') = (p' \circ \mathit{pretty}_{\mathit{fix}}\ p)\ (l, p\ r \mathbin{+\!\!+} cs)\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ cs'\end{aligned}$$

$= \{$Inductive hypothesis$\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l', cs') = \mathit{strictify}_{\mathit{fix}}\ s\ (l, p\ r \mathbin{+\!\!+} cs)\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ cs'\end{aligned}$$

$= \{\mathit{strictify}_{\mathit{fix}}\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l', cs') = l\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs)\\ &\quad\phantom{\textbf{where}}\ (t', \mathit{tTotal}) = s\ l\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ cs'\end{aligned}$$

$= \{$Simple case analysis on l ($\bot$ or not $\bot$), pattern matching$\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l', cs') = (l\ \text{`}\mathit{seq}\text{`}\ t', l\ \text{`}\mathit{seq}\text{`}\ \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs)\\ &\quad\phantom{\textbf{where}}\ (t', \mathit{tTotal}) = s\ l\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ cs'\end{aligned}$$

$= \{\mathit{seq}$, if $l = \bot$ then $t' = \mathit{tTotal} = \bot$ since s is strict$\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l', cs') = (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs)\\ &\quad\phantom{\textbf{where}}\ (t', \mathit{tTotal}) = s\ l\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ cs'\end{aligned}$$

$= \{$**where**$\}$

$$\begin{aligned}&(B\ t'\ r', cs'')\\ &\quad \textbf{where}\ (t', \mathit{tTotal}) = s\ l\\ &\quad\phantom{\textbf{where}}\ (r', cs'') = p'\ (\mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ p\ r \mathbin{+\!\!+} cs)\end{aligned}$$

$= \{$Rename variables$\}$

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

Before continuing we will prove that

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

$=$

$$\begin{aligned}&(B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}\ \text{`}\mathit{seq}\text{`}\ cs)\\ &\quad \textbf{where}\ (r', \mathit{rTotal}) = s\ r\end{aligned}$$

using case analysis on $lTotal$.

First case, $lTotal = \bot$:

$$\begin{aligned}&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') = p'\ (\bot\ \text{`}seq\text{`}\ p\ r \mathbin{+\!\!+} cs)\\=\ &\{seq\}\\&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') = p'\ \bot\\=\ &\{p' \text{ is strict}\}\\&(B\ l'\ \bot, \bot)\\=\ &\{seq,\ \textbf{where}\}\\&(B\ l'\ (\bot\ \text{`}seq\text{`}\ r'), \bot\ \text{`}seq\text{`}\ rTotal\ \text{`}seq\text{`}\ cs)\\&\quad \textbf{where}\ (r', rTotal) = s\ r\end{aligned}$$

Second case, $lTotal = ()$:

$$\begin{aligned}&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') = p'\ (()\ \text{`}seq\text{`}\ p\ r \mathbin{+\!\!+} cs)\\=\ &\left\{seq,\ pretty_{\mathit{fix}},\ \circ\right\}\\&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') = (p' \circ pretty_{\mathit{fix}}\ p)\ (r, cs)\\=\ &\{\text{Inductive hypothesis}\}\\&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') = strictify_{\mathit{fix}}\ s\ (r, cs)\\=\ &\left\{strictify_{\mathit{fix}}\right\}\\&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') \quad = r\ \text{`}seq\text{`}\ (t', tTotal\ \text{`}seq\text{`}\ cs)\\&\qquad\qquad\ (t', tTotal) = s\ r\\=\ &\{\text{We can remove } r\ \text{`}seq\text{`} \text{ using the same reasoning as for } l\ \text{`}seq\text{`} \text{ above}\}\\&(B\ l'\ r', cs'')\\&\quad \textbf{where}\ (r', cs'') \quad = (t', tTotal\ \text{`}seq\text{`}\ cs)\\&\qquad\qquad\ (t', tTotal) = s\ r\\=\ &\{\textbf{where}\}\\&(B\ l'\ t', tTotal\ \text{`}seq\text{`}\ cs)\\&\quad \textbf{where}\ (t', tTotal) = s\ r\\=\ &\{\text{Rename variables, } seq\}\\&(B\ l'\ (lTotal\ \text{`}seq\text{`}\ r'), ()\ \text{`}seq\text{`}\ rTotal\ \text{`}seq\text{`}\ cs)\\&\quad \textbf{where}\ (r', rTotal) = s\ r\end{aligned}$$

Now let us continue with the original proof:

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

$= \{$As shown above$\}$

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

$= \{$**where**, *seq* associative$\}$

$$\begin{array}{l}
(t', tTotal\ \text{`}seq\text{`}\ cs) \\
\quad \textbf{where} \\
\quad (t', tTotal) = (B\ l'\ (lTotal\ \text{`}seq\text{`}\ r'), lTotal\ \text{`}seq\text{`}\ rTotal) \\
\quad (l', lTotal) = s\ l \\
\quad (r', rTotal) = s\ r
\end{array}$$

$= \left\{strictify_{step},\ seq\right\}$

$$\begin{array}{l}
B\ l\ r\ \text{`}seq\text{`}\ (t', tTotal\ \text{`}seq\text{`}\ cs) \\
\quad \textbf{where}\ (t', tTotal) = strictify_{step}\ s\ (B\ l\ r)
\end{array}$$

$= \left\{strictify_{fix}\right\}$

$$strictify_{fix}\ (strictify_{step}\ s)\ (B\ l\ r, cs)$$

Hence we have shown $P_1(pretty_{step}\ p, parse_{step}\ p', strictify_{step}\ s)$, 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:

$$\forall m \in \mathbb{N}\,.\ \ pp_m \sqsubseteq id :: String \to String. \tag{12}$$

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

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

(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 [GH05, Sect. 4] is enough to verify this variant. We get that, for all $m \in \mathbb{N}$ and $cs :: String$,

$$\begin{gathered}
pp_m\ cs \sqsubseteq id\ cs \quad \text{iff}\\
\forall n \in Nat_{\text{fin}}\,.\ \ approx\ n\ (pp_m\ cs) \sqsubseteq approx\ n\ (id\ cs).
\end{gathered} \tag{13}$$

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

$$\begin{aligned}
&\forall n \in Nat_{\text{fin}}\,.\ \ \forall m \in \mathbb{N}\,.\ \ \forall cs :: String\,.\\
&\qquad\qquad\qquad approx\ n\ (pp_m\ cs) \sqsubseteq approx\ n\ (id\ cs),
\end{aligned} \tag{14}$$

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 (14) 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, $head\ cs \notin \{\texttt{'L'}, \texttt{'B'}\}$:

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

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

$$
\begin{aligned}
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pp}_m\ (\texttt{'L'} : cs')) \\
&= \{\mathit{pp}_m,\ m \geq 1\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ t_1 +\!\!+\ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+\ \mathit{pretty}'\ t_m +\!\!+\ cs_m) \\
& \qquad \textbf{where}\ (t_1,\ cs_1) = \mathit{parse}\ (\texttt{'L'} : cs') \\
& \qquad \phantom{\textbf{where}}\ (t_2,\ cs_2) = \mathit{parse}\ cs_1 \\
& \qquad \qquad \qquad \vdots \\
& \qquad \phantom{\textbf{where}}\ (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
&= \{\mathit{parse},\ \textbf{where},\ \text{note that if } m = 1 \text{ then } cs_m = cs'\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ L +\!\!+\ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+\ \mathit{pretty}'\ t_m +\!\!+\ cs_m) \\
& \qquad \textbf{where}\ (t_2,\ cs_2) = \mathit{parse}\ cs' \\
& \qquad \qquad \qquad \vdots \\
& \qquad \phantom{\textbf{where}}\ (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
&= \{\mathit{pretty}',\ +\!\!+\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\texttt{'L'} : \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+\ \mathit{pretty}'\ t_m +\!\!+\ cs_m) \\
& \qquad \textbf{where}\ (t_2,\ cs_2) = \mathit{parse}\ cs' \\
& \qquad \qquad \qquad \vdots \\
& \qquad \phantom{\textbf{where}}\ (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
&= \{\mathit{approx}\} \\
& \quad \texttt{'L'} : \mathit{approx}\ n\ (\mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+\ \mathit{pretty}'\ t_m +\!\!+\ cs_m) \\
& \qquad \textbf{where}\ (t_2,\ cs_2) = \mathit{parse}\ cs' \\
& \qquad \qquad \qquad \vdots \\
& \qquad \phantom{\textbf{where}}\ (t_m, cs_m) = \mathit{parse}\ cs_{m-1} \\
&= \big\{\mathit{pp}_{m-1},\ m \geq 1\big\} \\
& \quad \texttt{'L'} : \mathit{approx}\ n\ (\mathit{pp}_{m-1}\ cs') \\
&\sqsubseteq \{\text{Inductive hypothesis, monotonicity}\} \\
& \quad \texttt{'L'} : \mathit{approx}\ n\ (\mathit{id}\ cs') \\
&= \{\mathit{id},\ \mathit{approx}\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\texttt{'L'} : cs')
\end{aligned}
$$

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

$$
\begin{aligned}
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pp}_m\ (\texttt{'B'} : cs')) \\
&= \{\mathit{pp}_m,\ m \geq 1\}
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ t_1 +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ \mathit{cs}_m) \\
& \qquad \textbf{where}\ (t_1,\ \mathit{cs}_1) = \mathit{parse}\ (\texttt{'B'} : \mathit{cs}') \\
& \qquad\qquad\quad (t_2,\ \mathit{cs}_2) = \mathit{parse}\ \mathit{cs}_1 \\
& \qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\quad (t_m, \mathit{cs}_m) = \mathit{parse}\ \mathit{cs}_{m-1} \\
= \ & \{\mathit{parse}, \textbf{where}\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\mathit{pretty}'\ (B\ l\ r) +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ \mathit{cs}_m) \\
& \qquad \textbf{where}\ (l,\ \mathit{ls}) = \mathit{parse}\ \mathit{cs}' \\
& \qquad\qquad\quad (r,\ \mathit{rs}) = \mathit{parse}\ \mathit{ls} \\
& \qquad\qquad\quad (t_2,\ \mathit{cs}_2) = \mathit{parse}\ \mathit{rs} \\
& \qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\quad (t_m, \mathit{cs}_m) = \mathit{parse}\ \mathit{cs}_{m-1} \\
= \ & \{\mathit{pretty}', +\!\!+, +\!\!+ \text{ associative}\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n) \\
& \qquad (\texttt{'B'} : \mathit{pretty}'\ l +\!\!+ \mathit{pretty}'\ r +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ \mathit{cs}_m) \\
& \qquad \textbf{where}\ (l,\ \mathit{ls}) = \mathit{parse}\ \mathit{cs}' \\
& \qquad\qquad\quad (r,\ \mathit{rs}) = \mathit{parse}\ \mathit{ls} \\
& \qquad\qquad\quad (t_2,\ \mathit{cs}_2) = \mathit{parse}\ \mathit{rs} \\
& \qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\quad (t_m, \mathit{cs}_m) = \mathit{parse}\ \mathit{cs}_{m-1} \\
= \ & \{\mathit{approx}\} \\
& \quad \texttt{'B'} : \mathit{approx}\ n \\
& \qquad\qquad (\mathit{pretty}'\ l +\!\!+ \mathit{pretty}'\ r +\!\!+ \mathit{pretty}'\ t_2 +\!\!+ \ldots +\!\!+ \mathit{pretty}'\ t_m +\!\!+ \mathit{cs}_m) \\
& \qquad \textbf{where}\ (l,\ \mathit{ls}) = \mathit{parse}\ \mathit{cs}' \\
& \qquad\qquad\quad (r,\ \mathit{rs}) = \mathit{parse}\ \mathit{ls} \\
& \qquad\qquad\quad (t_2,\ \mathit{cs}_2) = \mathit{parse}\ \mathit{rs} \\
& \qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\quad (t_m, \mathit{cs}_m) = \mathit{parse}\ \mathit{cs}_{m-1} \\
= \ & \{\mathit{pp}_{m+1}\} \\
& \quad \texttt{'B'} : \mathit{approx}\ n\ (\mathit{pp}_{m+1}\ \mathit{cs}') \\
\sqsubseteq \ & \{\text{Inductive hypothesis, monotonicity}\} \\
& \quad \texttt{'B'} : \mathit{approx}\ n\ (\mathit{id}\ \mathit{cs}') \\
= \ & \{\mathit{id}, \mathit{approx}\} \\
& \quad \mathit{approx}\ (\mathit{Succ}\ n)\ (\texttt{'B'} : \mathit{cs}')
\end{aligned}
$$

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 $\mathit{parse} \circ \mathit{pretty} = \mathit{strictify} :: (T, \mathit{String}) \rightarrow (T, \mathit{String})$ 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:

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

Using this function we can formulate the approximation lemma as

$$\mathit{p1} = \mathit{p2} \quad \text{iff} \quad \forall n \in \mathit{Nat}_{\text{fin}}\,.\ \mathit{approxP}\ n\ \mathit{p1} = \mathit{approxP}\ n\ \mathit{p2} \qquad (15)$$

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 (14).) We prove the corresponding statement for $\mathit{Succ}\ n$ by case analysis on p.

First case, $p = \bot$:

$$\begin{array}{ll} & \mathit{approxP}\ (\mathit{Succ}\ n)\ ((\mathit{parse} \circ \mathit{pretty})\ \bot) \\ = & \{\circ,\ \mathit{pretty},\ \mathit{parse}\} \\ & \mathit{approxP}\ (\mathit{Succ}\ n)\ \bot :: (T, \mathit{String}) \\ = & \{\mathit{strictify}\} \\ & \mathit{approxP}\ (\mathit{Succ}\ n)\ (\mathit{strictify}\ \bot) :: (T, \mathit{String}) \end{array}$$

Second case, $p = (\bot, \mathit{cs})$ for an arbitrary $\mathit{cs} :: \mathit{String}$:

$$\begin{array}{ll} & \mathit{approxP}\ (\mathit{Succ}\ n)\ ((\mathit{parse} \circ \mathit{pretty})\ (\bot, \mathit{cs})) \\ = & \{\circ,\ \mathit{pretty},\ \mathit{pretty}',\ +\!\!+,\ \mathit{parse}\} \\ & \mathit{approxP}\ (\mathit{Succ}\ n)\ \bot :: (T, \mathit{String}) \\ = & \{\mathit{seq}\} \\ & \mathit{approxP}\ (\mathit{Succ}\ n)\ (\bot\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})) \\ & \quad \textbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ \bot \\ = & \{\mathit{strictify}\} \\ & \mathit{approxP}\ (\mathit{Succ}\ n)\ (\mathit{strictify}\ (\bot, \mathit{cs})) \end{array}$$

Third case, $p = (L, cs)$ for an arbitrary $cs :: String$:

$$
\begin{aligned}
& \quad approxP\ (Succ\ n)\ ((parse \circ pretty)\ (L, cs)) \\
= & \{\circ,\ pretty,\ pretty',\ +\!\!+,\ parse\} \\
& \quad approxP\ (Succ\ n)\ (L, cs) \\
= & \{seq\} \\
& \quad approxP\ (Succ\ n)\ (L, ()\ \text{`}seq\text{`}\ cs) \\
= & \{\textbf{where}\} \\
& \quad approxP\ (Succ\ n)\ (t', tTotal\ \text{`}seq\text{`}\ cs) \\
& \qquad \textbf{where}\ (t', tTotal) = (L, ()) \\
= & \{strictify'\} \\
& \quad approxP\ (Succ\ n)\ (t', tTotal\ \text{`}seq\text{`}\ cs) \\
& \qquad \textbf{where}\ (t', tTotal) = strictify'\ L \\
= & \{seq\} \\
& \quad approxP\ (Succ\ n)\ (L\ \text{`}seq\text{`}\ (t', tTotal\ \text{`}seq\text{`}\ cs)) \\
& \qquad \textbf{where}\ (t', tTotal) = strictify'\ L \\
= & \{strictify\} \\
& \quad approxP\ (Succ\ n)\ (strictify\ (L, cs))
\end{aligned}
$$

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

$$
\begin{aligned}
& \quad approxP\ (Succ\ n)\ ((parse \circ pretty)\ (B\ l\ r, cs)) \\
= & \{\circ,\ pretty,\ pretty',\ +\!\!+,\ +\!\!+\ \text{associative}\} \\
& \quad approxP\ (Succ\ n)\ (parse\ (\texttt{'B'} : pretty'\ l +\!\!+ pretty'\ r +\!\!+ cs)) \\
= & \{parse,\ pretty,\ \circ\} \\
& \quad approxP\ (Succ\ n)\ (B\ l'\ r', cs'') \\
& \qquad \textbf{where}\ (l', cs') = (parse \circ pretty)\ (l, pretty'\ r +\!\!+ cs) \\
& \qquad\qquad\quad (r', cs'') = parse\ cs' \\
= & \{approxP,\ approx\} \\
& \quad (B\ (approx\ n\ l')\ (approx\ n\ r'), cs'') \\
& \qquad \textbf{where}\ (l', cs') = (parse \circ pretty)\ (l, pretty'\ r +\!\!+ cs) \\
& \qquad\qquad\quad (r', cs'') = parse\ cs' \\
= & \{\text{Push } approx\ n \text{ through the pairs, turning it into } approxP\ n\} \\
& \quad (B\ l'\ r', cs'') \\
& \qquad \textbf{where}\ (l', cs') = approxP\ n\ ((parse \circ pretty)\ (l, pretty'\ r +\!\!+ cs)) \\
& \qquad\qquad\quad (r', cs'') = approxP\ n\ (parse\ cs')
\end{aligned}
$$

$= \{\text{Inductive hypothesis}\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l',\ cs') = \mathit{approxP}\ n\ (\mathit{strictify}\ (l, \mathit{pretty}'\ r \mathbin{+\!\!+} cs))\\ &\qquad\qquad (r', cs'') = \mathit{approxP}\ n\ (\mathit{parse}\ cs')\end{aligned}$$

$= \{\mathit{strictify}\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l',\ cs') = \mathit{approxP}\ n\ (l\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r \mathbin{+\!\!+} cs))\\ &\qquad\qquad (t',\ \mathit{tTotal}) = \mathit{strictify}'\ l\\ &\qquad\qquad (r', cs'') = \mathit{approxP}\ n\ (\mathit{parse}\ cs')\end{aligned}$$

The proof proceeds by case analysis on l. First case, $l = \bot$:

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l',\ cs') = \mathit{approxP}\ n\ (\bot\ \text{`}\mathit{seq}\text{`}\\ &\qquad\qquad\qquad\qquad (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r \mathbin{+\!\!+} cs))\\ &\qquad\qquad (t',\ \mathit{tTotal}) = \mathit{strictify}'\ \bot\\ &\qquad\qquad (r', cs'') = \mathit{approxP}\ n\ (\mathit{parse}\ cs')\end{aligned}$$

$= \{\mathit{seq},\ \mathit{approxP}\}$

$$\begin{aligned}&(B\ l'\ r', cs'')\\ &\quad \textbf{where}\ (l',\ cs') = \bot\\ &\qquad\qquad (r', cs'') = \mathit{approxP}\ n\ (\mathit{parse}\ cs')\end{aligned}$$

$= \{\text{Pattern matching, } \textbf{where}\}$

$$\begin{aligned}&(B\ \bot\ r', cs'')\\ &\quad \textbf{where}\ (r', cs'') = \mathit{approxP}\ n\ (\mathit{parse}\ \bot)\end{aligned}$$

$= \{\mathit{parse},\ \mathit{approxP}, \text{ pattern matching, } \textbf{where}\}$

$$(B\ \bot\ \bot, \bot) :: (T, \mathit{String})$$

$= \{\mathit{approx}\}$

$$(B\ (\mathit{approx}\ n\ \bot)\ (\mathit{approx}\ n\ \bot), \bot) :: (T, \mathit{String})$$

$= \{\mathit{approx},\ \mathit{approxP}\}$

$$\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ \bot\ \bot, \bot) :: (T, \mathit{String})$$

$= \{\mathit{seq},\ \textbf{where}\}$

$$\begin{aligned}&\mathit{approxP}\ (\mathit{Succ}\ n)\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ cs)\\ &\quad \textbf{where}\ (t',\ \mathit{tTotal}) = (B\ \bot\ (\bot\ \text{`}\mathit{seq}\text{`}\ r'), \bot\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\\ &\qquad\qquad (r', \mathit{rTotal}) = \mathit{strictify}'\ r\end{aligned}$$

$= \{\textbf{where}, \text{ pattern matching, } \mathit{strictify}'\}$

$$\begin{aligned}&\mathit{approxP}\ (\mathit{Succ}\ n)\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\&\quad \textbf{where}\ (t', \mathit{tTotal}) = (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\\&\qquad\qquad (l', \mathit{lTotal}) = \mathit{strictify}'\ \bot\\&\qquad\qquad (r', \mathit{rTotal}) = \mathit{strictify}'\ r\end{aligned}$$

$= \{\mathit{strictify}'\}$

$$\begin{aligned}&\mathit{approxP}\ (\mathit{Succ}\ n)\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\&\quad \textbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ (B\ \bot\ r)\end{aligned}$$

$= \{\mathit{seq}\}$

$$\begin{aligned}&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ \bot\ r\ \text{`}\mathit{seq}\text{`}\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}))\\&\quad \textbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ (B\ \bot\ r)\end{aligned}$$

$= \{\mathit{strictify}\}$

$$\mathit{approxP}\ (\mathit{Succ}\ n)\ (\mathit{strictify}\ (B\ \bot\ r, \mathit{cs}))$$

Second case, $l = L$:

$$\begin{aligned}&(B\ l'\ r', \mathit{cs}'')\\&\quad \textbf{where}\ (l', \mathit{cs}') \quad = \mathit{approxP}\ n\ (L\ \text{`}\mathit{seq}\text{`}\\&\qquad\qquad\qquad\qquad\qquad (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ \mathit{cs}))\\&\qquad\qquad (t', \mathit{tTotal}) = \mathit{strictify}'\ L\\&\qquad\qquad (r', \mathit{cs}'') \quad = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}')\end{aligned}$$

$= \{\mathit{seq},\ \mathit{strictify}',\ \textbf{where}\}$

$$\begin{aligned}&(B\ l'\ r', \mathit{cs}'')\\&\quad \textbf{where}\ (l', \mathit{cs}') = \mathit{approxP}\ n\ (L, ()\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ \mathit{cs})\\&\qquad\qquad (r', \mathit{cs}'') = \mathit{approxP}\ n\ (\mathit{parse}\ \mathit{cs}')\end{aligned}$$

$= \{\mathit{seq},\ \mathit{approxP},\ \textbf{where}\}$

$$\begin{aligned}&(B\ (\mathit{approx}\ n\ L)\ r', \mathit{cs}'')\\&\quad \textbf{where}\ (r', \mathit{cs}'') = \mathit{approxP}\ n\ (\mathit{parse}\ (\mathit{pretty}'\ r +\!\!+\ \mathit{cs}))\end{aligned}$$

$= \{\mathit{pretty},\ \circ\}$

$$\begin{aligned}&(B\ (\mathit{approx}\ n\ L)\ r', \mathit{cs}'')\\&\quad \textbf{where}\ (r', \mathit{cs}'') = \mathit{approxP}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ (r, \mathit{cs}))\end{aligned}$$

$= \{$Inductive hypothesis$\}$

$$\begin{aligned}&(B\ (\mathit{approx}\ n\ L)\ r', \mathit{cs}'')\\&\quad \textbf{where}\ (r', \mathit{cs}'') = \mathit{approxP}\ n\ (\mathit{strictify}\ (r, \mathit{cs}))\end{aligned}$$

$= \{\mathit{approxP}$ and approx are strict in the second argument$\}$

$$\begin{aligned}&(B\ (\mathit{approx}\ n\ L)\ (\mathit{approx}\ n\ r'), \mathit{cs}'')\\&\quad \textbf{where}\ (r', \mathit{cs}'') = \mathit{strictify}\ (r, \mathit{cs})\end{aligned}$$

$= \{\mathit{approx},\ \mathit{approxP}\}$

$approxP\ (Succ\ n)\ (B\ L\ r', cs'')$
  where $(r', cs'') = strictify\ (r, cs)$

$= \{strictify$, simple case analysis on r, pattern matching$\}$

$approxP\ (Succ\ n)\ (B\ L\ r', cs'')$
  where $(r', cs'') = (r\ `seq`\ t', r\ `seq`\ tTotal\ `seq`\ cs)$
    $(t', tTotal) = strictify'\ r$

$= \{$**where**$\}$

$approxP\ (Succ\ n)\ (B\ L\ (r\ `seq`\ t'), r\ `seq`\ tTotal\ `seq`\ cs)$
  where $(t', tTotal) = strictify'\ r$

$= \{$Rename variables$\}$

$approxP\ (Succ\ n)\ (B\ L\ (r\ `seq`\ r'), r\ `seq`\ rTotal\ `seq`\ cs)$
  where $(r', rTotal) = strictify'\ r$

$= \{seq$, $r = \bot$ implies that $r' = rTotal = \bot\}$

$approxP\ (Succ\ n)\ (B\ L\ r', rTotal\ `seq`\ cs)$
  where $(r', rTotal) = strictify'\ r$

$= \{$**where**, $seq\}$

$approxP\ (Succ\ n)\ (t', tTotal\ `seq`\ cs)$
  where $(t', tTotal) = (B\ L\ (()\ `seq`\ r'), ()\ `seq`\ rTotal)$
    $(r', rTotal) = strictify'\ r$

$= \{$**where**, $strictify'\}$

$approxP\ (Succ\ n)\ (t', tTotal\ `seq`\ cs)$
  where $(t', tTotal) = (B\ l'\ (lTotal\ `seq`\ r'), lTotal\ `seq`\ rTotal)$
    $(l', lTotal) = strictify'\ L$
    $(r', rTotal) = strictify'\ r$

$= \{seq,\ strictify'\}$

$approxP\ (Succ\ n)\ (B\ L\ r\ `seq`\ (t', tTotal\ `seq`\ cs))$
  where $(t', tTotal) = strictify'\ (B\ L\ r)$

$= \{strictify\}$

$approxP\ (Succ\ n)\ (strictify\ (B\ L\ r, cs))$

Last case, $l = B\ l_1\ r_1$ for arbitrary subtrees $l_1, r_1 :: T$:

$(B\ l'\ r', cs'')$
  where
  $(l', cs') = approxP\ n\ (B\ l_1\ r_1\ `seq`\ (t', tTotal\ `seq`\ pretty'\ r +\!\!+\ cs))$
  $(t', tTotal) = strictify'\ (B\ l_1\ r_1)$
  $(r', cs'') = approxP\ n\ (parse\ cs')$

$= \{seq,\ strictify'$, **where**$\}$

$$
\begin{array}{l}
(B\ l'\ r', cs'') \\
\quad \textbf{where} \\
\quad (l',\ cs') \quad = \mathit{approxP}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'), \\
\qquad\qquad\qquad\qquad\qquad (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ cs) \\
\quad (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\quad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\quad (r',\ cs'') \quad = \mathit{approxP}\ n\ (\mathit{parse}\ cs')
\end{array}
$$

$= \{\mathit{approxP}, \textbf{where}\}$

$$
\begin{array}{l}
(B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', cs'') \\
\quad \textbf{where} \\
\quad (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\quad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\quad (r',\ cs'') \quad = \mathit{approxP}\ n \\
\qquad\qquad\qquad (\mathit{parse}\ ((\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ cs))
\end{array}
$$

Now we have two cases, depending on whether $\mathit{lTotal}\,\text{`}\mathit{seq}\text{`}\ \mathit{rTotal}$, i.e. snd $(\mathit{strictify}'\ l_1)\ \text{`}\mathit{seq}\text{`}\ \mathit{snd}\ (\mathit{strictify}'\ r_1)$, equals $\bot$ or not.

First case, $\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal} = \bot$:

$$
\begin{array}{l}
(B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', cs'') \\
\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad (r',\ cs'') \quad = \mathit{approxP}\ n\ (\mathit{parse}\ (\bot\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ cs))
\end{array}
$$

$= \{\mathit{seq},\ \mathit{parse},\ \mathit{approxP}\}$

$$
\begin{array}{l}
(B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', cs'') \\
\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad (r',\ cs'') \quad = \bot
\end{array}
$$

$= \{\text{Pattern matching}, \textbf{where}\}$

$$
\begin{array}{l}
(B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ \bot, \bot) \\
\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1
\end{array}
$$

$= \{\mathit{approx}\}$

$$
\begin{array}{l}
(B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ (\mathit{approx}\ n\ \bot), \bot) \\
\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1
\end{array}
$$

$= \{\mathit{approxP}\}$

$$
\begin{array}{l}
\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'))\ \bot, \bot) \\
\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1
\end{array}
$$

$= \{\text{Rename variables}\}$

$$\begin{array}{l}
\quad \mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'))\ \bot, \bot) \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}') = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1
\end{array}$$

$= \{\mathit{seq},\ \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal} = \bot,\ \textbf{where}\}$

$$\begin{array}{l}
\quad \mathit{approxP}\ (\mathit{Succ}\ n) \\
\qquad\qquad (B\ (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'))\ ((\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}')\ \text{`}\mathit{seq}\text{`}\ r'), \\
\qquad\qquad\ ((\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}')\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}') = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{rTotal}) = \mathit{strictify}'\ r
\end{array}$$

The rest of this case is identical to the final steps of the next case.

Second case, $\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal} = () \neq \bot$:

$$\begin{array}{l}
\quad (B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{cs}'') \quad = \mathit{approxP}\ n\ (\mathit{parse}\ (()\ \text{`}\mathit{seq}\text{`}\ \mathit{pretty}'\ r +\!\!+\ \mathit{cs}))
\end{array}$$

$= \{\mathit{seq},\ \mathit{pretty},\ \circ\}$

$$\begin{array}{l}
\quad (B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{cs}'') \quad = \mathit{approxP}\ n\ ((\mathit{parse} \circ \mathit{pretty})\ (r, \mathit{cs}))
\end{array}$$

$= \{\text{Inductive hypothesis}\}$

$$\begin{array}{l}
\quad (B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{cs}'') \quad = \mathit{approxP}\ n\ (\mathit{strictify}\ (r, \mathit{cs}))
\end{array}$$

$= \{\text{Push } \mathit{approxP}\ n \text{ through the pair, turning it into } \mathit{approx}\ n\}$

$$\begin{array}{l}
\quad (B\ (\mathit{approx}\ n\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1')))\ (\mathit{approx}\ n\ r'), \mathit{cs}'') \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{cs}'') \quad = \mathit{strictify}\ \ (r, \mathit{cs})
\end{array}$$

$= \{\mathit{approx},\ \mathit{approxP}\}$

$$\begin{array}{l}
\quad \mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'))\ r', \mathit{cs}'') \\
\qquad \textbf{where}\ (l_1', \mathit{lTotal}) = \mathit{strictify}'\ l_1 \\
\qquad\qquad\quad\ (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1 \\
\qquad\qquad\quad\ (r', \mathit{cs}'') \quad = \mathit{strictify}\ \ (r, \mathit{cs})
\end{array}$$

$= \{\mathit{strictify},\ \text{simple case analysis on } r,\ \text{pattern matching}\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'))\ r', \mathit{cs}'')\\
&\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1\\
&\qquad\qquad (r',\ \mathit{cs}'') \quad = (r\ \text{`}\mathit{seq}\text{`}\ t', r\ \text{`}\mathit{seq}\text{`}\ \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\qquad\qquad (t',\ \mathit{tTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\textbf{where}\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r_1'))\ (r\ \text{`}\mathit{seq}\text{`}\ t'),\\
&\qquad\qquad\qquad\qquad r\ \text{`}\mathit{seq}\text{`}\ \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l_1',\ \mathit{lTotal}) = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}) = \mathit{strictify}'\ r_1\\
&\qquad\qquad (t',\ \mathit{tTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{$Rename variables$\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'))\ (r\ \text{`}\mathit{seq}\text{`}\ r'),\\
&\qquad\qquad\qquad\qquad r\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l_1',\ \mathit{lTotal}') = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1\\
&\qquad\qquad (r',\ \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\mathit{seq},\ r = \bot$ implies that $r' = \mathit{rTotal} = \bot\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'))\ r', \mathit{rTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l_1',\ \mathit{lTotal}') = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1\\
&\qquad\qquad (r',\ \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\mathit{seq},\ \mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}' \neq \bot,\ \mathit{seq}$ associative$\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\\
&\qquad\quad (B\ (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'))\ ((\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}')\ \text{`}\mathit{seq}\text{`}\ r'),\\
&\qquad\qquad ((\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}')\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l_1',\ \mathit{lTotal}') = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1\\
&\qquad\qquad (r',\ \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\textbf{where}\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l',\ \mathit{lTotal}) = (B\ l_1'\ (\mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ r_1'), \mathit{lTotal}'\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}')\\
&\qquad\qquad (l_1',\ \mathit{lTotal}') = \mathit{strictify}'\ l_1\\
&\qquad\qquad (r_1', \mathit{rTotal}') = \mathit{strictify}'\ r_1\\
&\qquad\qquad (r',\ \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\mathit{strictify}'\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal})\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})\\
&\quad \textbf{where}\ (l',\ \mathit{lTotal}) = \mathit{strictify}'\ (B\ l_1\ r_1)\\
&\qquad\qquad (r', \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\textbf{where}\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) \\
&\quad \textbf{where}\ (t', \mathit{tTotal}) = (B\ l'\ (\mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ r'), \mathit{lTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{rTotal}) \\
&\qquad\qquad (l', \mathit{lTotal}) = \mathit{strictify}'\ (B\ l_1\ r_1) \\
&\qquad\qquad (r', \mathit{rTotal}) = \mathit{strictify}'\ r
\end{aligned}$$

$= \{\mathit{strictify}'\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (t', \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs}) \\
&\quad \textbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ (B\ (B\ l_1\ r_1)\ r)
\end{aligned}$$

$= \{\mathit{seq}\}$

$$\begin{aligned}
&\mathit{approxP}\ (\mathit{Succ}\ n)\ (L\ \text{`}\mathit{seq}\text{`}\ (B\ (B\ l_1\ r_1)\ r, \mathit{tTotal}\ \text{`}\mathit{seq}\text{`}\ \mathit{cs})) \\
&\quad \textbf{where}\ (t', \mathit{tTotal}) = \mathit{strictify}'\ (B\ (B\ l_1\ r_1)\ r)
\end{aligned}$$

$= \{\mathit{strictify}\}$

$$\mathit{approxP}\ (\mathit{Succ}\ n)\ (\mathit{strictify}\ (B\ (B\ l_1\ r_1)\ r, \mathit{cs}))$$

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.

Testing 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 [CH00]. 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. (Our implementation can be downloaded from the first author's web page [Dan05].)

Comparing Proof Methods 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 (12), 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.

Other Proof Methods 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 [GH05], might belong to that category. We have proved (1’) using coinduction. That proof required a complex generalisation along the lines of (12). We have also tried to prove (1’) by using fusion. Due to the nature of the programs the standard fusion method seems inapplicable; a monadic variant is a better fit. The programs can be transformed into monadic forms (which of course carries extra proof obligations). Using fold fusion directly fails, but the more general method based on universal properties seems to work. The proof requires some rather complicated-looking lemmas which we have not verified, though. Hence this proof method seems to be the most complicated one to use for this particular case study. Furthermore the aim when using fusion is typically to *derive* the definition of (in this case) *strictify* from the definitions of *parse* and *pretty*, and the proof required some “eureka” steps which we doubt that we would have thought of if we had not known exactly what we aimed for.

The alert reader may have noticed that we almost proved (2) already in Sect. 4. We proved that the property was satisfied for all partial, finite input strings using strong induction. However, as may easily be verified, the predicate used was chain-complete. Hence, since every infinite string is the least upper bound of some chain of finite strings the property has to be true also for the infinite strings. Still, taking into account the lemma that we omitted, this proof is slightly more complicated than the one using fixpoint induction.

Why bother? 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') valid for all inputs were considerably longer than the ones for (1) limited to finite and total input, especially if one takes into account all work involved in rewriting the properties and programs. On the other hand, the proof of (2) using fixpoint induction was arguably easier than the one using ordinary (strong) induction. Still, it is not hard to see why people often ignore partial and infinite input.

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. In some cases 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.

Future Work 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

[C+03] 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/`.

[CH00] 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.

[Dan05] Nils Anders Danielsson. Personal web page, available at `http://www.cs.chalmers.se/~nad/`, 2005.

[Fax02] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.

[GH05] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. To appear in Fundamenta Informaticae Special Issue on Program Transformation, 2005.

[HG01] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, August 2001.

[HK] William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. Accepted for publication in the Journal of Functional Programming.

[HSH02] William Harrison, Tim Sheard, and James Hook. Fine control of demand in Haskell. In Eerke A. Boiten and Bernhard Möller, editors, *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 68–93, January 2002.

[Hug89] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[JJ02] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[JV04] 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.

[LPJ03] 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.

[PJ01] 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/`.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[San96] 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.

[Sch88] 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.

Fast and Loose Reasoning is Morally Correct*

Nils Anders Danielsson Jeremy Gibbons John Hughes

Abstract

We justify reasoning about non-total (partial) functional languages using methods seemingly only valid for total ones.

Two languages are defined, one total and one partial, with identical syntax. The semantics of the partial language includes partial and infinite values and lifted types, including lifted function spaces. A partial equivalence relation is then defined, the domain of which is the total subset of the partial language. It is proved that if two closed terms have the same semantics in the total language, then they have related semantics in the partial language.

1 Introduction

It is often claimed that functional programs are much easier to reason about than their imperative counterparts. Functional languages satisfy many pleasing equational laws, such as

$$\mathit{curry} \circ \mathit{uncurry} = \mathit{id}, \tag{1}$$
$$(\mathit{fst}\ x, \mathit{snd}\ x) = x \text{ and} \tag{2}$$
$$\mathit{fst}\ (x, y) = x, \tag{3}$$

and many others inspired by category theory. Such laws can be used to perform very pleasant proofs of program equality, and are indeed the foundation of an entire school of program transformation and derivation [Bir87, Mee86, Mal90, BdBH^{+}91, Jeu90]. There is just one problem. In real programming languages such as Haskell [PJ03] and ML [MTHM97], they do not hold.

The reason these laws fail is the presence of the undefined value $\perp$, and the fact that, in Haskell, $\perp$, $\lambda x.\perp$ and $(\perp, \perp)$ are all different (violating the

*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," and by the Royal Swedish Academy of Sciences.

first two laws above), while in ML, $\perp$, $(x, \perp)$ and $(\perp, y)$ are always the same (violating the third).

The fact that these laws are invalid does not prevent functional programmers from using them when developing programs, whether formally or informally. Squiggolers happily derive programs from specifications using them, and then transcribe the programs into Haskell in order to run them, confident that the programs will correctly implement the specification. Countless functional programmers happily curry or uncurry functions, confident that at worst they are changing definedness a little in obscure cases. Yet is this confidence justified? Reckless use of invalid laws can lead to patently absurd conclusions: for example, in ML, since $(x, \perp) = (y, \perp)$ for any x and y, we can use the third law above to conclude that $x = y$, for any x and y. How do we know that, when transforming programs using laws of this sort, we do not, for example, transform a correctly terminating program into an infinitely looping one?

This is the question we address in this paper. We call the unjustified reasoning with laws of this sort "fast and loose", and we show, under some mild and unsurprising conditions, that its conclusions are "morally correct". In particular, it is impossible to transform a terminating program into a looping one. Our results justify the hand reasoning that functional programmers already perform, and can be applied in proof checkers and automated provers to justify ignoring $\perp$-cases much of the time, thus simplifying proofs considerably.

In the next section we give an example showing how it can be burdensome to keep track of all preconditions when one is only interested in finite and total values, but is reasoning about a program written in a partial language. Section 3 is devoted to defining the language that we focus on, its syntax and two different semantics; one set-theoretic and one domain-theoretic. Section 4 briefly discusses partial equivalence relations (PERs), and Section 5 introduces a PER on the domain-theoretic semantics. This PER is used to model totality. In Section 6 a partial surjective homomorphism from the set-theoretic semantics to the quotient of the domain-theoretic semantics given by the PER is exhibited, and in Section 7 we use this homomorphism to prove our main result; fast and loose reasoning is morally correct. We go back to our earlier example and show how it fits in with the theory in Section 8. We also exhibit another example where reasoning directly about the partial language may be preferable (Section 9). Section 10 recasts the theory for a strict language, Section 11 discusses related work, and Section 12 concludes with a discussion of the results and possible future extensions of the theory.

Most proofs needed for the development below are only sketched; full proofs (around 80 pages) are available from Danielsson's web page [Dan05].

2 Propagating preconditions

Let us begin with an example. All code uses Haskell-like syntax, but some of it should be interpreted as being written in a total language.

First, let us say that we need to prove that the function $\mathit{map}\ (\lambda x.y+x) \circ \mathit{reverse} :: [\mathit{Nat}] \to [\mathit{Nat}]$ has a left inverse $\mathit{reverse} \circ \mathit{map}\ (\lambda x.x-y)$, perhaps as an intermediate step in some larger proof. (For a related example, see e.g. [Gib96].) In a total language we would do it more or less like this:

$$\begin{aligned}
& (\mathit{reverse} \circ \mathit{map}\ (\lambda x.x-y)) \circ (\mathit{map}\ (\lambda x.y+x) \circ \mathit{reverse}) \\
= \ & \{\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g),\ \circ \text{ associative}\} \\
& \mathit{reverse} \circ \mathit{map}\ ((\lambda x.x-y) \circ (\lambda x.y+x)) \circ \mathit{reverse} \\
= \ & \{(\lambda x.x-y) \circ (\lambda x.y+x) = \mathit{id}\} \\
& \mathit{reverse} \circ \mathit{map}\ \mathit{id} \circ \mathit{reverse} \\
= \ & \{\mathit{map}\ \mathit{id} = \mathit{id}\} \\
& \mathit{reverse} \circ \mathit{id} \circ \mathit{reverse} \\
= \ & \{\mathit{id} \circ f = f,\ \circ \text{ associative}\} \\
& \mathit{reverse} \circ \mathit{reverse} \\
= \ & \{\mathit{reverse} \circ \mathit{reverse} = \mathit{id}\} \\
& \mathit{id}.
\end{aligned}$$

Note the lemmas used for the proof, especially

$$(\lambda x.x-y) \circ (\lambda x.y+x) = \mathit{id} \tag{4}$$

and

$$\mathit{reverse} \circ \mathit{reverse} = \mathit{id}. \tag{5}$$

Consider now the task of repeating this proof in the context of some language based on partial functions, such as Haskell. To be concrete, let us assume that the natural number data type *Nat* is defined in the usual way,

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}. \tag{6}$$

Note that this type contains many properly partial values that do not correspond to any natural number, and also a total but infinite value. Let us also assume that $(+)$ and $(-)$ are defined by

$$(+) = \mathit{fold}\ \mathit{Succ} \tag{7}$$

and

$$(-) = \mathit{fold}\ \mathit{pred}, \tag{8}$$

where $\mathit{fold} :: (a \to a) \to a \to \mathit{Nat} \to a$ is the fold over natural numbers ($\mathit{fold}\ s\ z\ n$ replaces all occurrences of Succ in n with s, and Zero with z), and $\mathit{pred} :: \mathit{Nat} \to \mathit{Nat}$ is the predecessor function with $\mathit{pred}\ \mathit{Zero} = \mathit{Zero}$. The other functions and types are all standard [PJ03], which e.g. means that the list type also contains properly partial and infinite values.

Given these definitions the property proved above is no longer true. The proof breaks down in various places. More to the point, both lemmas (4) and (5) fail, and they fail due to both properly partial values,

$$(\mathit{Succ}\ \mathit{Zero} + \mathit{Succ}\ \bot) - \mathit{Succ}\ \mathit{Zero} = \bot \neq \mathit{Succ}\ \bot, \tag{9}$$
$$\mathit{reverse}\ (\mathit{reverse}\ (\mathit{Zero} : \bot)) = \bot \neq \mathit{Zero} : \bot, \tag{10}$$

and infinite ones,

$$(\mathit{fix}\ \mathit{Succ} + \mathit{Zero}) - \mathit{fix}\ \mathit{Succ} = \bot \neq \mathit{Zero}, \tag{11}$$
$$\mathit{reverse}\ (\mathit{reverse}\ (\mathit{repeat}\ \mathit{Zero})) = \bot \neq \mathit{repeat}\ \mathit{Zero}. \tag{12}$$

(Here fix is the fixpoint combinator, i.e. $\mathit{fix}\ \mathit{Succ}$ is the "infinite" lazy natural number. The application $\mathit{repeat}\ x$ yields an infinite list containing only x.) Note that (4) would also have failed if we had used a fixed-precision integer type with overflow control. In fact $\mathit{id} \circ f = f$ also fails since we have lifted function spaces and $\mathit{id} \circ \bot = \lambda x.\bot \neq \bot$, but that does not affect this example since $\mathit{reverse} \neq \bot$.

These problems are not surprising, they are the price you pay for partiality. Values that are properly partial and/or infinite have different properties than their total, finite counterparts. A reasonable solution is to restrict ourselves to total, finite values. Let us see what the proof looks like then. We have to η-expand our property, and assume that $\mathit{xs} :: [\mathit{Nat}]$ is a total, finite list and that $y :: \mathit{Nat}$ is a total, finite natural number. (Note the terminology used here: if a list is said to be total, then all elements in the list are assumed to be total as well, and similarly for finite lists. The concepts of totality and finiteness are discussed in more detail in Sections 5.3 and 8, respectively.) We get

$$\begin{aligned} & ((\mathit{reverse} \circ \mathit{map}\ (\lambda x.x - y)) \circ (\mathit{map}\ (\lambda x.y + x) \circ \mathit{reverse}))\ \mathit{xs} \\ = \ & \{\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g), \text{definition of } \circ\} \end{aligned}$$

$$
\begin{aligned}
& \mathit{reverse}\ (\mathit{map}\ ((\lambda x.x - y) \circ (\lambda x.y + x))\ (\mathit{reverse}\ \mathit{xs})) \\
= & \left\{ \begin{array}{l} \bullet\ \mathit{map}\ f\ \mathit{xs} = \mathit{map}\ g\ \mathit{xs} \text{ if } \mathit{xs} \text{ is total and } f\ x = g\ x \text{ for all total } x, \\ \bullet\ \mathit{reverse}\ \mathit{xs} \text{ is total, finite if } \mathit{xs} \text{ is,} \\ \bullet\ ((\lambda x.x - y) \circ (\lambda x.y + x))\ x = \mathit{id}\ x \text{ for total } x \text{ and total, finite } y \end{array} \right\} \\
& \mathit{reverse}\ (\mathit{map}\ \mathit{id}\ (\mathit{reverse}\ \mathit{xs})) \\
= & \{\mathit{map}\ \mathit{id} = \mathit{id}\} \\
& \mathit{reverse}\ (\mathit{id}\ (\mathit{reverse}\ \mathit{xs})) \\
= & \{\text{definition of } \mathit{id}\} \\
& \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \\
= & \{\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs} \text{ for total, finite } \mathit{xs}\} \\
& \mathit{xs}.
\end{aligned}
$$

All steps are more or less identical, using similar lemmas, except for the second step, where two new lemmas are required. How come that step becomes so unwieldy? The problem seems to be that, although we know that $((\lambda x.x - y) \circ (\lambda x.y + x)) = \mathit{id}$ given total input, and also that xs only contains total natural numbers, we have to *propagate* this precondition through *reverse* and *map*.

One way of looking at the problem is that the type system used is too weak. If there were a type for total, finite natural numbers, and similarly for lists, then the propagation would be handled by the types of *reverse* and *map*. The imaginary total language used for the first proof effectively has such a type system.

Note that the two versions of the program are written using identical syntax, and the semantic rules for the total language and the partial language are probably more or less identical when only total, finite (or even infinite) values are considered. Does not this imply that we can get the second result above, with all preconditions, by using the first proof? The answer is more or less yes, and proving this is what many of the sections below will be devoted to. In Section 8 we come back to this example and spell out in full detail what "more or less" means in this case.

3 Language

Now we will define the main language discussed in the text. It is a strongly typed, monomorphic functional language with recursive (polynomial) types and their corresponding fold and unfold operators. Having only folds and unfolds is not a serious limitation; it is e.g. easy to implement primitive recursion over lists or natural numbers inside the language.

Since we want our results to be applicable to reasoning about Haskell programs we include the explicit strictness operator seq, which forces us to have lifted function spaces in the domain-theoretic semantics given below. The semantics of seq is defined in Figure 6. We discuss the most important differences between Haskell and this language in Section 12.

The term syntax of the language, $\mathcal{L}_1$, is inductively defined as

$$\begin{aligned} t ::= {} & x \mid t_1\ t_2 \mid \lambda x.t \\ & \mid \mathsf{seq} \mid \star \\ & \mid (,) \mid \mathsf{fst} \mid \mathsf{snd} \\ & \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \\ & \mid \mathsf{in}_F \mid \mathsf{out}_F \mid \mathsf{fold}_F \mid \mathsf{unfold}_F. \end{aligned} \tag{13}$$

The pairing function (,) can be used in a distfix style, as in (t_1, t_2). The type syntax is defined by

$$\sigma,\ \tau,\ \gamma ::= \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau \mid 1 \mid \mu F \mid \nu F \tag{14}$$

and

$$F,\ G ::= \mathit{Id} \mid K_\sigma \mid F \times G \mid F + G. \tag{15}$$

The letters F and G range over functors; Id is the identity functor and K_σ is the constant functor with $K_\sigma\ \tau = \sigma$ (informally). The functor indices on in, out, fold and unfold are sometimes omitted below. The types μF and νF are inductive and coinductive types, respectively. As an example $\mu(K_1 + \mathit{Id})$ is the type of finite natural numbers, and $\nu(K_1 + \mathit{Id})$ is the type of natural numbers extended with infinity. The type constructor $\to$ is sometimes used right associatively, without explicit parentheses.

In order to discuss general recursion we define the language $\mathcal{L}_2$ to be $\mathcal{L}_1$ extended with

$$t ::= \dots \mid \mathsf{fix}. \tag{16}$$

However, whenever fix is not explicitly mentioned, the language discussed is $\mathcal{L}_1$ (or one of the restrictions of $\mathcal{L}_1$ introduced below).

We only consider well-typed terms according to the typing rules in Figure 1. To ease the presentation we also introduce some syntactic sugar for terms and types, see Figures 2 and 3.

Two different denotational semantics are defined for these languages, one domain-theoretic ($[\![\cdot]\!]$, modelled on languages like Haskell) and one set-theoretic ($\langle\!\langle\cdot\rangle\!\rangle$, modelled on total languages). The semantic domains for all

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\begin{array}{ll}
\Gamma \vdash \mathsf{seq} : \sigma \to \tau \to \tau & \Gamma \vdash \mathsf{fix} : (\sigma \to \sigma) \to \sigma \\
\Gamma \vdash \star : 1 & \Gamma \vdash (,) : \sigma \to \tau \to (\sigma \times \tau) \\
\Gamma \vdash \mathsf{fst} : (\sigma \times \tau) \to \sigma & \Gamma \vdash \mathsf{snd} : (\sigma \times \tau) \to \tau \\
\Gamma \vdash \mathsf{inl} : \sigma \to (\sigma + \tau) & \Gamma \vdash \mathsf{inr} : \tau \to (\sigma + \tau) \\
\multicolumn{2}{l}{\Gamma \vdash \mathsf{case} : (\sigma + \tau) \to (\sigma \to \gamma) \to (\tau \to \gamma) \to \gamma} \\
\Gamma \vdash \mathsf{in}_F : F\ \mu F \to \mu F & \Gamma \vdash \mathsf{out}_F : \nu F \to F\ \nu F \\
\Gamma \vdash \mathsf{fold}_F : (F\ \sigma \to \sigma) \to \mu F \to \sigma & \Gamma \vdash \mathsf{unfold}_F : (\sigma \to F\ \sigma) \to \sigma \to \nu F
\end{array}$$

Figure 1: Typing rules for $\mathcal{L}_1$ and $\mathcal{L}_2$.

$$\begin{aligned}
\circ &\mapsto \lambda f\, g\, x.f\ (g\ x) \\
Id &\mapsto \lambda f\, x.f\ x \\
K_\sigma &\mapsto \lambda f\, x.x \\
F \times G &\mapsto \lambda f\, x.\mathsf{seq}\ x\ (F\ f\ (\mathsf{fst}\ x), G\ f\ (\mathsf{snd}\ x)) \\
F + G &\mapsto \lambda f\, x.\mathsf{case}\ x\ (\mathsf{inl} \circ F\ f)\ (\mathsf{inr} \circ G\ f)
\end{aligned}$$

Figure 2: Syntactic sugar for terms.

$$\begin{aligned}
Id\ \sigma &\mapsto \sigma \\
K_\tau\ \sigma &\mapsto \tau \\
(F \times G)\ \sigma &\mapsto F\ \sigma \times G\ \sigma \\
(F + G)\ \sigma &\mapsto F\ \sigma + G\ \sigma
\end{aligned}$$

Figure 3: Syntactic sugar for types.

types are defined in Figure 4. We define the semantics of recursive types by appealing to category-theoretic work [BdM96, FM91]. For instance, the set-theoretic semantic domain of μF is the codomain of the initial object in F-$\mathsf{Alg}(\mathsf{SET})$. Here SET is the category of sets and total functions, and F-$\mathsf{Alg}(\mathsf{SET})$ is the category of F-algebras (in SET) and homomorphisms between them. The initial object (which is known to exist given our limitations on F) is a function $in_F : F\ \mu F \to \mu F$ (note the codomain). Initiality implies that for any function $f : F\ A \to A$ there is a unique function $fold_F\ f : \mu F \to A$, satisfying the universal property

$$\forall h : \mu F \to A. \qquad h = fold_F\ f \ \Leftrightarrow\ h \circ in_F = f \circ F\ h. \tag{17}$$

This is how $\langle\!\langle \mathsf{fold} \rangle\!\rangle$ is defined. To define $\langle\!\langle \mathsf{unfold} \rangle\!\rangle$ we go via the *final* object in F-$\mathsf{Coalg}(\mathsf{SET})$ (the category of F-coalgebras) instead. The semantics of all terms are given in Figure 6.

The domain-theoretic semantics lives in the category CPO of pointed complete partial orders (CPOs) and continuous functions. To define $[\![\mu F]\!]$ the category $\mathsf{CPO}_\perp$ of CPOs and *strict* continuous functions is also used. We want all types in the domain-theoretic semantics to be lifted (like in Haskell). To model this we lift all functors using $L(\cdot)$, which is defined in Figure 5.

If we were to define $[\![\mathsf{fold}]\!]$ using the same method as for $\langle\!\langle \mathsf{fold} \rangle\!\rangle$, then that would restrict its arguments to be strict functions. An explicit fixpoint is used instead. The construction still satisfies the universal property associated with *fold* if all functions involved are strict. For symmetry we also define $[\![\mathsf{unfold}]\!]$ using an explicit fixpoint; that does not affect its universality property.

We have been a little sloppy above; we have not defined the action of the functor K_σ on objects. When working in SET we let $K_\sigma\ A = \langle\!\langle \sigma \rangle\!\rangle$, and in CPO and $\mathsf{CPO}_\perp$ we let $K_\sigma\ A = [\![\sigma]\!]$. Otherwise the functors have their usual meanings.

Note that in_F is only defined at type $F\ \mu F \to \mu F$, and out_F only at $\nu F \to F\ \nu F$. However, in_F has an inverse which we (ambiguously) denote by out_F, and similarly for out_F. We have not included these inverses in the term language, but that is not a fundamental limitation since $out_F \in [\![\mu F \to F\ \mu F]\!]$ and $in_F \in [\![F\ \nu F \to \nu F]\!]$ are given by $[\![\mathsf{fold}\ \ (F\ \mathsf{in}_F)]\!]$ and $[\![\mathsf{unfold}\ \ (F\ \mathsf{out}_F)]\!]$, respectively (and similarly for $\langle\!\langle \cdot \rangle\!\rangle$).

The semantics of fix is, as usual, given by a least fixpoint construction. We cannot perform such a construction in SET, so the set-theoretic semantics (which is supposed to model a total language anyway) is only given for $\mathcal{L}_1$; $\langle\!\langle \mathsf{fix} \rangle\!\rangle$ is not defined.

Some notes about the notation used: When t is closed we sometimes use $[\![t]\!]$ as a shorthand for $[\![t]\!]\ \rho$, and similarly for $\langle\!\langle \cdot \rangle\!\rangle$. On the semantic side $\cdot_\perp$ is

$$\begin{aligned}
[\![\sigma \to \tau]\!] &= \langle [\![\sigma]\!] \to [\![\tau]\!] \rangle_\perp & \langle\!\langle \sigma \to \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle \to \langle\!\langle \tau \rangle\!\rangle \\
[\![\sigma \times \tau]\!] &= ([\![\sigma]\!] \times [\![\tau]\!])_\perp & \langle\!\langle \sigma \times \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle \times \langle\!\langle \tau \rangle\!\rangle \\
[\![\sigma + \tau]\!] &= ([\![\sigma]\!] + [\![\tau]\!])_\perp & \langle\!\langle \sigma + \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle + \langle\!\langle \tau \rangle\!\rangle \\
[\![1]\!] &= 1_\perp & \langle\!\langle 1 \rangle\!\rangle &= 1
\end{aligned}$$

$$\begin{aligned}
[\![\mu F]\!] &= \text{The codomain of the initial object in } L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\perp). \\
\langle\!\langle \mu F \rangle\!\rangle &= \text{The codomain of the initial object in } F\text{-}\mathsf{Alg}(\mathsf{SET}). \\
[\![\nu F]\!] &= \text{The domain of the final object in } L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO}). \\
\langle\!\langle \nu F \rangle\!\rangle &= \text{The domain of the final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET}).
\end{aligned}$$

Figure 4: Semantic domains for types.

$$\begin{aligned}
L(Id) &= Id \\
L(K_\sigma) &= K_\sigma \\
L(F \times G) &= (L(F) \times L(G))_\perp \\
L(F + G) &= (L(F) + L(G))_\perp
\end{aligned}$$

Figure 5: Lifting of functors.

the lifting operator, and $\langle \cdot \to \cdot \rangle$ is the continuous function space constructor. Furthermore $\times$ is cartesian product, and $+$ is separated sum; $A + B$ contains elements of the form $inl(a)$ with $a \in A$ and $inr(b)$ with $b \in B$. The one-point CPO/set is denoted by 1, with $\star$ as the only element. Since we use lifted function spaces, we use special notation for lifted function application,

$$f@x = \begin{cases} \perp, & f = \perp, \\ f\ x, & \text{otherwise.} \end{cases} \tag{18}$$

Many functions used on the meta-level are not lifted, though, so @ is not used very much below. Finally note that we are a little sloppy below, in that we do not write out liftings explicitly; we write (x, y) for a non-bottom element of $(A \times B)_\perp$, for instance.

$$
\begin{array}{rlrl}
[\![x]\!]\,\rho &= \rho(x) & \langle\!\langle x\rangle\!\rangle\,\rho &= \rho(x)\\
[\![t_1 t_2]\!]\,\rho &= ([\![t_1]\!]\,\rho)\,@\,([\![t_2]\!]\,\rho) & \langle\!\langle t_1 t_2\rangle\!\rangle\,\rho &= (\langle\!\langle t_1\rangle\!\rangle\,\rho)\;\;(\langle\!\langle t_2\rangle\!\rangle\,\rho)\\
[\![\lambda x.t]\!]\,\rho &= \lambda v.\,[\![t]\!]\,\rho[x\mapsto v] & \langle\!\langle \lambda x.t\rangle\!\rangle\,\rho &= \lambda v.\,\langle\!\langle t\rangle\!\rangle\,\rho[x\mapsto v]\\
[\![\mathsf{seq}]\!] &= \lambda v_1\,v_2.\begin{cases}\bot, & v_1=\bot\\ v_2, & \text{otherwise}\end{cases} & \langle\!\langle \mathsf{seq}\rangle\!\rangle &= \lambda v_1\,v_2.v_2\\
[\![\mathsf{fix}]\!] &= \lambda f.\displaystyle\bigsqcup_{i=0}^{\infty} f^i@\bot & \langle\!\langle \mathsf{fix}\rangle\!\rangle & \text{is not defined.}\\
[\![\star]\!] &= \star & \langle\!\langle \star\rangle\!\rangle &= \star\\
[\![(,)]\!] &= \lambda v_1\,v_2.(v_1,v_2) & \langle\!\langle (,)\rangle\!\rangle &= \lambda v_1\,v_2.(v_1,v_2)\\
[\![\mathsf{fst}]\!] &= \lambda v.\begin{cases}\bot, & v=\bot\\ v_1, & v=(v_1,v_2)\end{cases} & \langle\!\langle \mathsf{fst}\rangle\!\rangle &= \lambda(v_1,v_2).v_1\\
[\![\mathsf{snd}]\!] &= \lambda v.\begin{cases}\bot, & v=\bot\\ v_2, & v=(v_1,v_2)\end{cases} & \langle\!\langle \mathsf{snd}\rangle\!\rangle &= \lambda(v_1,v_2).v_2\\
[\![\mathsf{inl}]\!] &= \lambda v.\mathit{inl}(v) & \langle\!\langle \mathsf{inl}\rangle\!\rangle &= \lambda v.\mathit{inr}(v)\\
[\![\mathsf{inr}]\!] &= \lambda v.\mathit{inl}(v) & \langle\!\langle \mathsf{inr}\rangle\!\rangle &= \lambda v.\mathit{inr}(v)\\
[\![\mathsf{case}]\!] &= \lambda v\,f_1\,f_2.\begin{cases}\bot, & v=\bot\\ f_1@v_1, & v=\mathit{inl}(v_1)\\ f_2@v_2, & v=\mathit{inr}(v_2)\end{cases} & \langle\!\langle \mathsf{case}\rangle\!\rangle &= \lambda v\,f_1\,f_2.\begin{cases}f_1\;v_1, & v=\mathit{inl}(v_1)\\ f_2\;v_2, & v=\mathit{inr}(v_2)\end{cases}
\end{array}
$$

$$
\begin{aligned}
[\![\mathsf{in}_F]\!] &= \begin{cases}\text{The initial object in } L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\bot)\text{, viewed as a morphism}\\ \text{in } \mathsf{CPO}.\end{cases}\\
\langle\!\langle \mathsf{in}_F\rangle\!\rangle &= \begin{cases}\text{The initial object in } F\text{-}\mathsf{Alg}(\mathsf{SET})\text{, viewed as a morphism in}\\ \mathsf{SET}.\end{cases}\\
[\![\mathsf{out}_F]\!] &= \begin{cases}\text{The final object in } L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO})\text{, viewed as a morphism}\\ \text{in } \mathsf{CPO}.\end{cases}\\
\langle\!\langle \mathsf{out}_F\rangle\!\rangle &= \begin{cases}\text{The final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET})\text{, viewed as a morphism in}\\ \mathsf{SET}.\end{cases}\\
[\![\mathsf{fold}_F]\!] &= \lambda f.\,[\![\mathsf{fix}]\!]\;\;(\lambda g.f\circ[\![F]\!]\;\;g\circ[\![\mathsf{out}]\!])\\
\langle\!\langle \mathsf{fold}_F\rangle\!\rangle &= \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Alg}(\mathsf{SET}) \text{ from } \langle\!\langle \mathsf{in}\rangle\!\rangle \text{ to } f,\\ \text{viewed as a morphism in } \mathsf{SET}.\end{cases}\\
[\![\mathsf{unfold}_F]\!] &= \lambda f.\,[\![\mathsf{fix}]\!]\;\;(\lambda g.\,[\![\mathsf{in}]\!]\circ[\![F]\!]\;\;g\circ f)\\
\langle\!\langle \mathsf{unfold}_F\rangle\!\rangle &= \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Coalg}(\mathsf{SET}) \text{ from } f \text{ to } \langle\!\langle \mathit{out}\rangle\!\rangle,\\ \text{viewed as a morphism in } \mathsf{SET}.\end{cases}
\end{aligned}
$$

Figure 6: Semantics of well-typed terms, for some context ρ mapping variables to semantic values.

4 Partial equivalence relations

In what follows we will use *partial equivalence relations*, or PERs for short.

A PER on a set S is a symmetric and transitive binary relation on S. For a PER R on S, and some $x \in S$ with xRx, define the *equivalence class* of x as

$$[x]_R = \{\, y \mid y \in S,\ xRy \,\}. \tag{19}$$

(The index R is omitted below.) Note that the equivalence classes partition $\mathrm{dom}(R) = \{\, x \in S \mid xRx \,\}$, the *domain* of R. Let $[R]$ denote the set of equivalence classes of R.

For convenience we will use the notation $\{c\}$ for an arbitrary element $x \in c$, where c is an equivalence class of some PER $R \subseteq S^2$. This definition is of course ambiguous, but the ambiguity disappears in many contexts. Given the PER defined in Section 5 we have that $[\mathit{inl}(\{c\})]$ denotes the same equivalence class no matter which element in c is chosen, for instance.

5 Moral equality

We will now inductively define a family of PERs $\sim_\sigma \subseteq [\![\sigma]\!]^2$ on the domain-theoretic semantic domains. (The index σ will sometimes be omitted.) The intention is that if σ does not contain function spaces, then we should have $x \sim_\sigma y$ iff x and y are total, equal values. For functions we will have $f \sim g$ iff f and g map (total) related values to related values. If two values x and y are related by $\sim$, then we say that they are *morally equal*.

Given this family of PERs we can relate the set-theoretic semantic values with the total values of the domain-theoretic semantics, see Sections 6 and 7.

5.1 Non-recursive types

The PER $\sim$ is a logical relation, i.e. we have the following definition for function spaces,

$$\begin{aligned} f \sim_{\sigma\to\tau} g \ \Leftrightarrow\ & f \neq \bot \ \wedge\ g \neq \bot \ \wedge \\ & \forall x, y \in [\![\sigma]\!] .\ x \sim_\sigma y \ \Rightarrow\ f@x \sim_\tau g@y. \end{aligned} \tag{20}$$

We need to ensure that f and g are non-bottom since some of the PERs will turn out to have $\bot \in \mathrm{dom}(\sim)$.

Pairs are related if corresponding components are related,

$$\begin{aligned} x \sim_{\sigma\times\tau} y \ \Leftrightarrow\ & \exists x_1, y_1 \in [\![\sigma]\!] ,\ x_2, y_2 \in [\![\tau]\!] . \\ & x = (x_1, x_2) \ \wedge\ y = (y_1, y_2) \ \wedge\ x_1 \sim_\sigma y_1 \ \wedge\ x_2 \sim_\tau y_2. \end{aligned} \tag{21}$$

Similarly, sums are related if they are of the same kind with related components,

$$\begin{aligned} x \sim_{\sigma+\tau} y \;\Leftrightarrow\;& (\exists x', y' \in [\![\sigma]\!] \,.\; x = \mathit{inl}(x') \;\wedge\; y = \mathit{inl}(y') \;\wedge\; x' \sim_\sigma y') \\ &\vee\; (\exists x', y' \in [\![\tau]\!] \,.\; x = \mathit{inr}(x') \;\wedge\; y = \mathit{inr}(y') \;\wedge\; x' \sim_\tau y') \,. \end{aligned} \tag{22}$$

The value $\star$ of the unit type is related to itself and $\bot$ is not related to anything,

$$x \sim_1 y \;\Leftrightarrow\; x = y = \star. \tag{23}$$

It is easy to check that what we have so far yields a PER.

5.2 Recursive types

The definition for recursive types is trickier. Consider lists. When should a list be related to another? Given the intentions above it seems reasonable for *xs* to be related to *ys* whenever they have the same, total list structure (spine), and elements at corresponding positions are recursively related. In other words, something like

$$\begin{aligned} &\mathit{xs} \sim_{\mu(K_1+(K_\sigma\times \mathit{Id}))} \mathit{ys} \;\Leftrightarrow \\ &\qquad (\mathit{xs} = \mathit{in}\ \mathit{inl}(\star) \;\wedge\; \mathit{ys} = \mathit{in}\ \mathit{inl}(\star)) \\ &\qquad\quad \vee \Big(\exists x, y \in [\![\sigma]\!] \,,\; \mathit{xs}', \mathit{ys}' \in [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] \,. \\ &\qquad\qquad \mathit{xs} = \mathit{in}\ \mathit{inr}((x, \mathit{xs}')) \;\wedge\; \mathit{ys} = \mathit{in}\ \mathit{inr}((y, \mathit{ys}')) \\ &\qquad\qquad \wedge\, x \sim_\sigma y \;\wedge\; \mathit{xs}' \sim_{\mu(K_1+(K_\sigma\times \mathit{Id}))} \mathit{ys}'\Big). \end{aligned} \tag{24}$$

There is one problem with this definition; it is recursive, so it is not entirely clear what it means.

We formalise the intuition embodied in (24) by defining an operator $O(F)$ for each functor F,

$$\begin{aligned} &O(F) : \wp\big([\![\mu F]\!]^2\big) \to \wp\big([\![\mu F]\!]^2\big) \\ &O(F)(X) = \{\, (\mathit{in}\ a, \mathit{in}\ b) \mid (a, b) \in O'_F(F)(X) \,\} \end{aligned} \tag{25}$$

(where $\wp(X)$ is the power set of X). The helper $O'_F(G)$ is defined by

$$\begin{aligned} &O'_F(G) : \wp\big([\![\mu F]\!]^2\big) \to \wp\big([\![G\ \mu F]\!]^2\big) \\ &O'_F(\mathit{Id})(X) \qquad = X \\ &O'_F(K_\sigma)(X) \qquad = \{\, (x, y) \mid x, y \in [\![\sigma]\!] \,,\; x \sim y \,\} \\ &O'_F(F_1 \times F_2)(X) = \left\{\, ((a_1, b_1), (a_2, b_2)) \;\middle|\; \begin{matrix} (a_1, a_2) \in O'_F(F_1)(X), \\ (b_1, b_2) \in O'_F(F_2)(X) \end{matrix} \,\right\} \\ &O'_F(F_1 + F_2)(X) = \{\, (\mathit{inl}(x') \,, \mathit{inl}(y')) \mid (x', y') \in O'_F(F_1)(X) \,\} \cup \\ &\qquad\qquad\qquad\quad\; \{\, (\mathit{inr}(x') \,, \mathit{inr}(y')) \mid (x', y') \in O'_F(F_2)(X) \,\} \,. \end{aligned} \tag{26}$$

The operators are defined for inductive types. However, replacing μF with νF in the definitions is enough to yield operators suitable for coinductive types.

Now, note that $O(F)$ is a monotone operator on the complete lattice $\wp\big(\llbracket \mu F \rrbracket^2\big)$. This implies that it has both least and greatest fixpoints [Pri02], which leads to the following definitions,

$$\begin{aligned} x \sim_{\mu F} y \;&\Leftrightarrow\; (x,y) \in \mu O(F) \;\text{ and} &(27)\\ x \sim_{\nu F} y \;&\Leftrightarrow\; (x,y) \in \nu O(F)\,. &(28)\end{aligned}$$

These definitions may not be entirely transparent. If we go back to the list example and expand the definition of $O(K_1 + (K_\sigma \times \mathit{Id}))$ we get

$$\begin{aligned} &O(K_1 + (K_\sigma \times \mathit{Id}))(X) = \\ &\quad \{\, (\mathit{in}\ \mathit{inl}(\star)\,,\mathit{in}\ \mathit{inl}(\star))\,\} \cup \\ &\quad \left\{ \begin{matrix} (\mathit{in}\ \mathit{inr}((x,\mathit{xs}))\,, \\ \mathit{in}\ \mathit{inr}((y,\mathit{ys}))) \end{matrix} \;\middle|\; x,y \in \llbracket\sigma\rrbracket\,,\ x \sim y,\ (\mathit{xs},\mathit{ys}) \in X \right\}. \end{aligned} \tag{29}$$

The least and greatest fixpoints of this operator correspond to our original aims for $\sim_{\mu(K_1+(K_\sigma\times \mathit{Id}))}$ and $\sim_{\nu(K_1+(K_\sigma\times \mathit{Id}))}$.

It is still possible to show that what we have defined is actually a PER, but it takes a little more work. First note the two proof principles given by the definitions above: induction,

$$\forall X \subseteq \llbracket \mu F\rrbracket^2\,.\ O(F)(X) \subseteq X \;\Rightarrow\; \mu O(F) \subseteq X, \tag{30}$$

and coinduction,

$$\forall X \subseteq \llbracket \nu F\rrbracket^2\,.\ X \subseteq O(F)(X) \;\Rightarrow\; X \subseteq \nu O(F)\,. \tag{31}$$

Many proofs needed for this text proceed according to the following scheme, named IIICI below (Induction-Induction-Induction-Coinduction-Induction):

- First induction over the type structure.
- For inductive types, induction according to (30) and then induction over the functor structure.
- For coinductive types, coinduction according to (31) and then induction over the functor structure.

Using this scheme we have proved that $\sim$ is a PER.

5.3 Properties

We can prove that $\sim$ satisfies a number of properties. Before we leave the subject of recursive types, we note that

$$in\ x \sim_{\mu F} in\ y \;\Leftrightarrow\; x \sim_{F\,\mu F} y \tag{32}$$

and

$$x \sim_{\nu F} y \;\Leftrightarrow\; out\ x \sim_{F\,\nu F} out\ y \tag{33}$$

hold, as well as the symmetric statements where μF is replaced by νF and vice versa. This is proved using a method similar to IIICI, but not quite identical.

Another method similar to IIICI can be used to verify that $\sim$ satisfies one of our initial goals: if σ does not contain function spaces, then $x \sim_\sigma y$ iff $x, y \in \mathrm{dom}(\sim_\sigma)$ and $x = y$. In fact, the goal was that x and y should be related iff they were equal, *total* values. That is easily arranged, though. We *define* the set of total values of type τ to be $\mathrm{dom}(\sim_\tau)$. This should correspond to basic intuition about totality for non-strict functional languages. Note that sometimes a definition of totality is used where $f \in [\![\sigma \to \tau]\!]$ is total iff $f\ x = \bot$ implies that $x = \bot$. That definition is not suitable for non-strict languages where most semantic domains are not flat. As a simple example, consider $[\![\mathsf{fst}]\!]$; we have $[\![\mathsf{fst}]\!] \in \mathrm{dom}(\sim)$, so $[\![\mathsf{fst}]\!]$ is total according to our definition, but $[\![\mathsf{fst}]\!]\ (\bot, \bot) = \bot$.

It can be proved by induction over the type structure that $\sim_\sigma$ is monotone when seen as a function $\sim_\sigma\colon [\![\sigma]\!]^2 \to 1_\bot$. We also have (by induction over the type structure) that $\bot \notin \mathrm{dom}(\sim_\sigma)$ for almost all types σ. The only exceptions are given by the grammar

$$\chi ::= \nu Id \mid \mu K_\chi \mid \nu K_\chi. \tag{34}$$

Note that $[\![\chi]\!] = \{\, \bot \,\}$ for all these types. The (near-complete) absence of bottoms in $\mathrm{dom}(\sim)$ gives us an easy way of showing that related values are not always equal: $[\![\mathsf{seq}]\!] \sim [\![\lambda x.\lambda y.y]\!]$ (at most types) but $[\![\mathsf{seq}]\!] \neq [\![\lambda x.\lambda y.y]\!]$. Since this example breaks down when seq is used at type $\chi \to \sigma \to \sigma$, let $\mathcal{L}_1'$ denote the language consisting of all terms from $\mathcal{L}_1$ which do not contain such uses of seq.

By using induction over the term structure instead of the type structure, and then follow the rest of IIICI, one can prove that the fundamental theorem of logical relations holds for any term t in $\mathcal{L}_1'$: if $\rho(x) \sim \rho'(x)$ for all free variables x in a term t, then

$$[\![t]\!]\,\rho \sim [\![t]\!]\,\rho'. \tag{35}$$

The fundamental theorem is important because it implies that $[\![t]\!] \in \mathrm{dom}(\sim_\sigma)$ for all closed terms $t : \sigma$ in $\mathcal{L}_1'$. Note, however, that $[\![\mathsf{fix}]\!] \notin \mathrm{dom}(\sim)$ (at most types) since $[\![\lambda x.x]\!] \in \mathrm{dom}(\sim)$ and $[\![\mathsf{fix}\ (\lambda x.x)]\!] = \bot$.

6 Partial surjective homomorphism

With the PER defined we can now prove that there is a *partial surjective homomorphism* [Fri75] from $\langle\!\langle\cdot\rangle\!\rangle$ to $\mathrm{dom}(\sim)$. This means that there is a partial, surjective function $j_\sigma : \langle\!\langle\sigma\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_\sigma]$ for each type σ, which for function types satisfies

$$j_{\tau_1 \to \tau_2}\ f\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x) \tag{36}$$

whenever $f \in \mathrm{dom}(j_{\tau_1 \to \tau_2})$ and $x \in \mathrm{dom}(j_{\tau_1})$. (Here $\tilde{\to}$ is the partial function space constructor and $\mathrm{dom}(f)$ denotes the domain of the partial function f.)

However, we need one restriction on the types allowed: for reasons detailed below we must avoid types σ with $\langle\!\langle\sigma\rangle\!\rangle = \emptyset$. This can be accomplished by restricting ourselves to types that syntactically contain 1, such as $1 \times \mu Id$ or $\nu\,(Id + K_1)$ (proof by induction over type structure). Note that none of the types χ introduced in Section 5 contain 1. Hence, let us define a further (final) restricted language $\mathcal{L}_1''$: $t \in \mathcal{L}_1''$ iff each subterm of t (including t itself) has a type that syntactically contains 1.

One can easily extend j to a partial functor

from the category which has types σ as objects and total functions between the corresponding set-theoretic domains $\langle\!\langle\sigma\rangle\!\rangle$ as morphisms

to the category which has types σ as objects and quotients of continuous functions between the corresponding domain-theoretic domains $[\![\sigma]\!]$ as morphisms.

The object part of the functor is the identity. The morphism part is given by j.

The functions $j_\sigma : \langle\!\langle\sigma\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_\sigma]$ are simultaneously proved to be well-defined and surjective by induction over the type structure plus some other techniques mentioned further down. The following basic cases are easy,

$$\begin{aligned} &j_{\sigma\times\tau} : \langle\!\langle\sigma \times \tau\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\sigma\times\tau}] \\ &j_{\sigma\times\tau}\ (x, y) = [(\{j_\sigma\ x\}, \{j_\tau\ y\})]\,, \end{aligned} \tag{37}$$

$$\begin{aligned} &j_{\sigma+\tau} : \langle\!\langle\sigma + \tau\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\sigma+\tau}] \\ &j_{\sigma+\tau}\ inl(x)\ = [inl(\{j_\sigma\ x\})] \\ &j_{\sigma+\tau}\ inr(y)\ = [inr(\{j_\tau\ y\})]\,, \text{ and} \end{aligned} \tag{38}$$

$$\begin{aligned} &j_1 : \langle\!\langle 1 \rangle\!\rangle \mathrel{\tilde{\to}} [\sim_1] \\ &j_1 \star = [\star] \, . \end{aligned} \tag{39}$$

Note the use of $\{\cdot\}$ to ease the description of these functions.

It turns out to be hard to come up with a total definition of j for function spaces; this is why our definition of j is partial. To define the function $j_{\tau_1 \to \tau_2} : \langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\tau_1 \to \tau_2}]$ we employ a technique originating from Friedman [Fri75]: if possible, let $j_{\tau_1 \to \tau_2}\ f$ be the element $g \in [\sim_{\tau_1 \to \tau_2}]$ satisfying

$$\forall x \in \mathrm{dom}(j_{\tau_1}) \, . \;\; g\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x). \tag{40}$$

If a g exists, then it can be shown to be unique (using surjectivity of j_{τ_1}). If no such g exists, then let $j_{\tau_1 \to \tau_2}\ f$ be undefined. To show that $j_{\tau_1 \to \tau_2}$ is surjective we need the assumption $\langle\!\langle \tau_2 \rangle\!\rangle \neq \emptyset$ mentioned above.[1] Note that this definition makes it easy to prove that j is homomorphic.

For inductive types we use the following definition of $j_{\mu F}$,

$$\begin{aligned} &j_{\mu F} : \langle\!\langle \mu F \rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\mu F}] \\ &j_{\mu F}\ x = [\mathit{in}\ \ \{J_{F,F}\ (\mathit{out}\ x)\}] \, , \end{aligned} \tag{41}$$

where

$$\begin{aligned} &J_{F,G} : \langle\!\langle G\ \mu F \rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{G\ \mu F}] \\ &J_{F,\mathit{Id}}\ x && = j_{\mu F}\ x \\ &J_{F,K_\sigma}\ x && = j_\sigma\ x \\ &J_{F,G_1 \times G_2}\ (x, y) && = [(\{J_{F,G_1}\ x\} , \{J_{F,G_2}\ y\})] \\ &J_{F,G_1 + G_2}\ \mathit{inl}(x) && = [\mathit{inl}(\{J_{F,G_1}\ x\})] \\ &J_{F,G_1 + G_2}\ \mathit{inr}(y) && = [\mathit{inr}(\{J_{F,G_2}\ y\})] \, . \end{aligned} \tag{42}$$

Note the recursive invocation of $j_{\mu F}$ above. For this to be well-defined we extend the induction used to prove that j is well-defined and surjective to lexicographic induction on first the type and then the *size* of values of inductive type. This size can be defined using the *fold* operator in the category SET.

Finally we define j for coinductive types. Since some coinductive values are infinite we cannot use simple recursion like for inductive types. Instead we define a total helper function using *unfold* from the category CPO directly,

$$\begin{aligned} &j'_{\nu F} : \langle \langle\!\langle \nu F \rangle\!\rangle_\perp \to [\![\nu F]\!] \rangle \\ &j'_{\nu F} = \mathit{unfold}\ \left(J'_{F,F} \circ \mathit{out} \right) , \end{aligned} \tag{43}$$

[1]Hence our restrictions on $\mathcal{L}''_1$ are slightly stronger than necessary.

and then wrap up the result whenever possible,

$$\begin{aligned} &j_{\nu F} : \langle\!\langle \nu F \rangle\!\rangle \tilde{\to} [\sim_{\nu F}] \\ &j_{\nu F}\ x = \begin{cases} [j'_{\nu F}\ x]\,, & j'_{\nu F}\ x \in \mathrm{dom}(\sim_{\nu F}) \\ \text{undefined}, & \text{otherwise.} \end{cases} \end{aligned} \tag{44}$$

Above we use $J'_{F,G}$, defined by

$$\begin{aligned} &J'_{F,G} : \langle \langle\!\langle G\ \nu F \rangle\!\rangle_{\perp} \to L(G)\ \langle\!\langle \nu F \rangle\!\rangle_{\perp} \rangle \\ &J'_{F,G}\ \bot && = \bot \\ &J'_{F,Id}\ x && = x \\ &J'_{F,K_\sigma}\ x && = \begin{cases} \{j_\sigma\ x\}\,, & x \in \mathrm{dom}(j_\sigma) \\ \bot, & \text{otherwise} \end{cases} \\ &J'_{F,G_1 \times G_2}\ (x, y) && = (J'_{F,G_1}\ x, J'_{F,G_2}\ y) \\ &J'_{F,G_1 + G_2}\ \mathit{inl}(x) && = \mathit{inl}\bigl(J'_{F,G_1}\ x\bigr) \\ &J'_{F,G_1 + G_2}\ \mathit{inr}(y) && = \mathit{inr}\bigl(J'_{F,G_2}\ y\bigr)\,. \end{aligned} \tag{45}$$

It is straightforward to check that this is a sound definition. Checking surjectivity requires more work. A somewhat subtle application of the generic approximation lemma [HG01] does the trick, though.

Given the definition of $j_{\sigma\to\tau}$ we easily get that $j\ \mathit{id} = [\mathit{id}]$ and, whenever $j_{\tau\to\gamma}\ f$ and $j_{\sigma\to\tau}\ g$ are defined, we have

$$j_{\sigma\to\gamma}\ (f \circ g) = j_{\tau\to\gamma}\ f \circ j_{\sigma\to\tau}\ g \tag{46}$$

(where $[f] \circ [g] = [f \circ g]$). Hence, as claimed, j is a partial functor between the categories mentioned above.

7 Main theorem

Now we get to our main theorem. Assume that t is a term in $\mathcal{L}''_1$ with contexts ρ and ρ' satisfying

$$\rho(x) \in \mathrm{dom}(\sim)\ \wedge\ j\ \rho'(x) = [\rho(x)] \tag{47}$$

for all variables x free in t. Then we have that $j\ (\langle\!\langle t \rangle\!\rangle\ \rho')$ is well-defined and

$$j\ (\langle\!\langle t \rangle\!\rangle\ \rho') = [[\![t]\!]\ \rho]\,. \tag{48}$$

This result can be proved by induction over the structure of t, induction over the size of values of inductive type and coinduction for coinductive types. The case where t is an application relies heavily on j being homomorphic.

As a corollary to the main theorem we get, for any two terms $t_1,\ t_2 : \sigma$ in $\mathcal{L}_1''$ with two pairs of contexts ρ_1, ρ_1' and ρ_2, ρ_2' both satisfying the conditions of (47) (for t_1 and t_2, respectively), that

$$\langle\!\langle t_1 \rangle\!\rangle\, \rho_1' = \langle\!\langle t_2 \rangle\!\rangle\, \rho_2' \;\Rightarrow\; [\![t_1]\!]\, \rho_1 \sim [\![t_2]\!]\, \rho_2. \tag{49}$$

In other words, if we can prove that two terms are equal in the world of sets, then they are morally equal in the world of domains. When formalised like this the reasoning performed using set-theoretic methods, "fast and loose" reasoning, is not loose any more.

8 Review of example

After having introduced the main theoretic body, let us now revisit the example from Section 2.

We verified that $\mathit{revMap} = \mathit{reverse} \circ \mathit{map}\ (\lambda x.x - y)$ is the left inverse of $\mathit{mapRev} = \mathit{map}\ (\lambda x.y + x) \circ \mathit{reverse}$ in a total setting. Let us express this result using the language introduced in Section 3. The types of the functions become $\mathit{ListNat} \to \mathit{ListNat}$, where $\mathit{ListNat}$ is the inductive type $\mu(K_1 + (K_{\mathit{Nat}} \times \mathit{Id}))$ of lists of natural numbers, and Nat is the inductive type $\mu(K_1 + \mathit{Id})$. Note also that the functions *reverse* and *map* can be expressed using folds, so the terms belong to $\mathcal{L}_1$. Finally note that all types used contain 1, so the terms belong to $\mathcal{L}_1''$, and we can make full use of the theory.

Our earlier proof in effect showed that

$$\langle\!\langle \mathit{revMap} \circ \mathit{mapRev} \rangle\!\rangle\ [y \mapsto y'] = \langle\!\langle \mathit{id} \rangle\!\rangle \tag{50}$$

for an arbitrary $y' \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$, which by (49) implies that

$$[\![\mathit{revMap} \circ \mathit{mapRev}]\!]\ [y \mapsto y''] \sim_{\mathit{ListNat} \to \mathit{ListNat}} [\![\mathit{id}]\!] \tag{51}$$

whenever $y'' \in \mathrm{dom}(\sim_{\mathit{Nat}})$ and $[y''] = j\ y'$ for some $y' \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$. By the fundamental theorem (35) and the main theorem (48) we have that $[\![t]\!]$ satisfies the conditions for y'' for any closed term $t \in \mathcal{L}_1''$ of type Nat. This includes all total, finite natural numbers.

It remains to interpret $\sim_{\mathit{ListNat} \to \mathit{ListNat}}$. Denote the left hand side of (51) by f. Since $f \neq \bot$ and $\mathit{id} \neq \bot$ the equation implies that $f @ \mathit{xs} \sim_{\mathit{ListNat}} \mathit{ys}$ whenever $\mathit{xs} \sim_{\mathit{ListNat}} \mathit{ys}$. By using the fact (mentioned in Section 5.3) that $\mathit{xs} \sim_{\mathit{ListNat}} \mathit{ys}$ iff $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ and $\mathit{xs} = \mathit{ys}$ we can restate the equation as $f@xs = xs$ whenever $xs \in \mathrm{dom}(\sim_{ListNat})$. Hence we have arrived at the statement proved by the more elaborate proof in Section 2, if $xs \in \mathrm{dom}(\sim_{ListNat})$ means the same as "*xs* is total and finite".

We defined totality to mean "related according to $\sim$" in Section 5.3, so by definition *xs* is total. We have not defined finiteness, though. However, with any reasonable definition we can be certain that $x \in \mathrm{dom}(\sim_\sigma)$ is finite if σ does not contain function spaces or coinductive types; in the absence of such types we can define a function $size_\sigma \in \mathrm{dom}(\sim_\sigma) \to \mathbb{N}$ which has the property that $size\ x' < size\ x$ whenever x' is a structurally smaller part of x, such as with $x = inl(x')$ or $x = in\ (x', x'')$. In fact, we used a similar function in order to justify the definition of $j_{\mu F}$ (41).

9 Partial reasoning is sometimes preferable

This section discusses an example of a different kind from the one given in Section 2; an example where partial reasoning (i.e. reasoning using the domain-theoretic semantics directly) seems to be more efficient than total reasoning.

We define two functions *sums* and *diffs*, both of type $ListNat \to ListNat$, with the inductive types *ListNat* and *Nat* defined just like in Section 8. The function *sums* takes a list of numbers and calculates their running sum, and *diffs* performs the left inverse operation, along the lines of

$$sums\ [3,1,4,1,5] = [3,4,8,9,14] \text{ and} \tag{52}$$

$$diffs\ [3,4,8,9,14] = [3,1,4,1,5]. \tag{53}$$

(Using standard syntactic sugar for lists and natural numbers.) The aim is to prove that

$$\langle\!\langle diffs \circ sums \rangle\!\rangle = id. \tag{54}$$

We do that in Section 9.1. Alternatively we can implement the functions in $\mathcal{L}_2$ and prove that

$$[\![diffs \circ sums]\!]\ xs = xs \tag{55}$$

for all total, finite lists $xs \in [\![ListNat]\!]$ containing total, finite natural numbers. That is done in Section 9.2. We then compare the experiences in Section 9.3.

9.1 Using total reasoning

First let us implement the functions in $\mathcal{L}_1''$. To make the development easier to follow we use some syntax borrowed from Haskell. We also use the function *foldr*, modelled on its Haskell namesake:

$$\begin{aligned}&\mathit{foldr} : (\sigma \to (\tau \to \tau)) \to \tau \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \to \tau\\ &\mathit{foldr}\ f\ x = \mathsf{fold}_{K_1+(K_\sigma\times \mathit{Id})}\ (\lambda y.\mathsf{case}\ y\ (\lambda \_.x)\ (\lambda p.f\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p)))\,.\end{aligned} \tag{56}$$

The following is a simple, albeit inefficient, recursive implementation of *sums*:

$$\begin{aligned}&\mathit{sums} : \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{sums} = \mathit{foldr}\ \mathit{add}\ [\,],\end{aligned} \tag{57}$$

where

$$\begin{aligned}&\mathit{add} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{add}\ x\ \mathit{ys} = x : \mathit{map}\ (\lambda y.x + y)\ \mathit{ys}.\end{aligned} \tag{58}$$

Here $(+)$ and $(-)$ (used below) are implemented as folds, in a manner analogous to (7) and (8) in Section 2. The function *map* can be implemented using *foldr*,

$$\begin{aligned}&\mathit{map} : (\sigma \to \tau) \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \to \mu(K_1 + (K_\tau \times \mathit{Id}))\\ &\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ \mathit{ys}.f\ x : \mathit{ys})\ [\,].\end{aligned} \tag{59}$$

The definition of *diffs* uses similar techniques:

$$\begin{aligned}&\mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{diffs} = \mathit{foldr}\ \mathit{sub}\ [\,],\end{aligned} \tag{60}$$

where

$$\begin{aligned}&\mathit{sub} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{sub}\ x\ \mathit{ys} = x : \mathit{toHead}\ (\lambda y.y - x)\ \mathit{ys}.\end{aligned} \tag{61}$$

The helper function *toHead* applies a function to the first element of a non-empty list, and leaves empty lists unchanged:

$$\begin{aligned}&\mathit{toHead} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{toHead}\ f\ (y : \mathit{ys}) = f\ y : \mathit{ys}\\ &\mathit{toHead}\ f\ [\,] \qquad\quad = [\,].\end{aligned} \tag{62}$$

Now let us prove (54). We can use fold fusion [BdM96],

$$\begin{aligned} & g \circ \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e' \\ & \quad \Leftarrow\ \forall x, y.\ g\ (f\ x\ y) = f'\ x\ (g\ y)\ \wedge\ g\ e = e'. \end{aligned} \tag{63}$$

(For simplicity we do not write out the semantic brackets $\langle\!\langle \cdot \rangle\!\rangle$, or any contexts.) We have

$$\begin{aligned} & \mathit{diffs} \circ \mathit{sums} = \mathit{id} \\ \Leftrightarrow\ & \{\text{definition of } \mathit{sums},\ \mathit{id} = \mathit{foldr}\ (:)\ [\,]\} \\ & \mathit{diffs} \circ \mathit{foldr}\ \mathit{add}\ [\,] = \mathit{foldr}\ (:)\ [\,] \\ \Leftarrow\ & \{\text{fold fusion}\} \\ & \mathit{diffs}\ [\,] = [\,]\ \wedge \\ & \forall x, \mathit{ys}.\ \mathit{diffs}\ (\mathit{add}\ x\ \mathit{ys}) = x : \mathit{diffs}\ \mathit{ys}. \end{aligned}$$

The first conjunct is trivial; for the second we have

$$\begin{aligned} & \mathit{diffs}\ (\mathit{add}\ x\ \mathit{ys}) \\ =\ & \{\text{definition of } \mathit{add}\} \\ & \mathit{diffs}\ (x : \mathit{map}\ (\lambda y.x + y)\ \mathit{ys}) \\ =\ & \{\text{definition of } \mathit{diffs}, \text{ semantics of } \mathit{foldr}\} \\ & \mathit{sub}\ x\ (\mathit{diffs}\ (\mathit{map}\ (\lambda y.x + y)\ \mathit{ys})) \\ =\ & \{\text{definition of } \mathit{sub}\} \\ & x : \mathit{toHead}\ (\lambda y.y - x)\ (\mathit{diffs}\ (\mathit{map}\ (\lambda y.x + y)\ \mathit{ys})) \\ =\ & \{\text{lemma (see below), definition of } \circ\} \\ & x : \mathit{toHead}\ (\lambda y.y - x)\ (\mathit{toHead}\ (\lambda y.x + y)\ (\mathit{diffs}\ \mathit{ys})) \\ =\ & \{\mathit{toHead}\ f \circ \mathit{toHead}\ g = \mathit{toHead}\ (f \circ g), \text{ definition of } \circ\} \\ & x : \mathit{toHead}\ ((\lambda y.y - x) \circ (\lambda y.x + y))\ (\mathit{diffs}\ \mathit{ys}) \\ =\ & \{(\lambda y.y - x) \circ (\lambda y.x + y) = \mathit{id}\} \\ & x : \mathit{toHead}\ \mathit{id}\ (\mathit{diffs}\ \mathit{ys}) \\ =\ & \{\mathit{toHead}\ \mathit{id} = \mathit{id}\} \\ & x : \mathit{id}\ (\mathit{diffs}\ \mathit{ys}) \\ =\ & \{\text{definition of } \mathit{id}\} \\ & x : \mathit{diffs}\ \mathit{ys}. \end{aligned}$$

To prove the lemma

$$\mathit{diffs} \circ \mathit{map}\ (\lambda y.x + y) = \mathit{toHead}\ (\lambda y.x + y) \circ \mathit{diffs} \tag{64}$$

we use fold-map fusion [BdM96],

$$foldr\ f\ e \circ map\ g = foldr\ (f \circ g)\ e. \tag{65}$$

We have

$$\begin{aligned}
& \mathit{diffs} \circ map\ (\lambda y.x+y) \\
= \ & \{\text{definition of } \mathit{diffs}\} \\
& foldr\ sub\ [\,] \circ map\ (\lambda y.x+y) \\
= \ & \{\text{fold-map fusion}\} \\
& foldr\ (sub \circ (\lambda y.x+y))\ [\,] \\
= \ & \{\text{fold fusion, see below}\} \\
& toHead\ (\lambda y.x+y) \circ foldr\ sub\ [\,] \\
= \ & \{\text{definition of } \mathit{diffs}\} \\
& toHead\ (\lambda y.x+y) \circ \mathit{diffs}.
\end{aligned}$$

To finish up we have to verify that the preconditions for fold fusion are satisfied above,

$$toHead\ (\lambda y.x+y)\ [\,] = [\,] \tag{66}$$

and

$$\begin{aligned}
\forall y, ys.\ \ & toHead\ (\lambda y.x+y)\ (sub\ y\ ys) = \\
& (sub \circ (\lambda y.x+y))\ y\ (toHead\ (\lambda y.x+y)\ ys).
\end{aligned} \tag{67}$$

The first one is yet again trivial; for the second one we have

$$\begin{aligned}
& toHead\ (\lambda y.x+y)\ (sub\ y\ ys) \\
= \ & \{\text{definition of } sub\} \\
& toHead\ (\lambda y.x+y)\ (y : toHead\ (\lambda z.z-y)\ ys) \\
= \ & \{\text{definition of } toHead\} \\
& x+y : toHead\ (\lambda z.z-y)\ ys \\
= \ & \{\lambda z.z-y = (\lambda z.z-(x+y)) \circ (\lambda y.x+y)\} \\
& x+y : toHead\ ((\lambda z.z-(x+y)) \circ (\lambda y.x+y))\ ys \\
= \ & \{toHead\ f \circ toHead\ g = toHead\ (f \circ g)\ , \text{definition of } \circ\} \\
& x+y : toHead\ (\lambda z.z-(x+y))\ (toHead\ (\lambda y.x+y)\ ys) \\
= \ & \{\text{definition of } sub\} \\
& sub\ (x+y)\ (toHead\ (\lambda y.x+y)\ ys) \\
= \ & \{\text{definition of } \circ\} \\
& (sub \circ (\lambda y.x+y))\ y\ (toHead\ (\lambda y.x+y)\ ys).
\end{aligned}$$

9.2 Using partial reasoning

Let us now see what we can accomplish when we are not restricted to the limitations of a total language. Yet again we borrow some syntax from Haskell; most notably we do not use fix directly, but define functions using recursive equations instead.

The definitions above used structural recursion. The programs below instead use structural corecursion, as captured by the function *unfoldr*, which is based on the standard unfold for lists as given by the Haskell Report [PJ03]:

$$\begin{aligned} &\mathit{unfoldr} : (\tau \to (1 + (\sigma \times \tau))) \to \tau \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \\ &\mathit{unfoldr}\ f\ b = \mathsf{case}\ (f\ b)\ (\lambda\_.[\,])\ (\lambda p.\mathsf{fst}\ p : \mathit{unfoldr}\ f\ (\mathsf{snd}\ p)). \end{aligned} \tag{68}$$

Note that we cannot use unfold here, since it has the wrong type. We can write *unfoldr* with the aid of fix, though. In total languages inductive and coinductive types cannot easily be mixed; we do not have the same problem in partial languages.

The corecursive definition of *sums*,

$$\begin{aligned} &\mathit{sums} : \mathit{ListNat} \to \mathit{ListNat} \\ &\mathit{sums}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{next}\ (0, \mathit{xs}), \end{aligned} \tag{69}$$

with helper *next*,

$$\begin{aligned} &\mathit{next} : (\mathit{Nat} \times \mathit{ListNat}) \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat}))) \\ &\mathit{next}\ (e, [\,]) \qquad = \mathsf{inl}\ \star \\ &\mathit{next}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (e + x, (e + x, \mathit{xs})), \end{aligned} \tag{70}$$

should be just as easy to follow as the recursive one, if not easier. Here we have used the same definitions of $(+)$ and $(-)$ as above, and 0 is shorthand for in (inl $\star$). The definition of *diffs*,

$$\begin{aligned} &\mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat} \\ &\mathit{diffs}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{step}\ (0, \mathit{xs}), \end{aligned} \tag{71}$$

with *step*,

$$\begin{aligned} &\mathit{step} : (\mathit{Nat} \times \mathit{ListNat}) \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat})) \\ &\mathit{step}\ (e, [\,]) \qquad = \mathsf{inl}\ \star \\ &\mathit{step}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (x - e, (x, \mathit{xs})), \end{aligned} \tag{72}$$

is arguably more natural than the previous one.

Now we can prove (55) for all total lists containing total, finite natural numbers; we do not need to restrict ourselves to finite lists. To do that we use the approximation lemma [HG01],

$$xs = ys \quad \Leftrightarrow \quad \forall n \in \mathbb{N}.\ \mathit{approx}\ n\ xs = \mathit{approx}\ n\ ys, \tag{73}$$

where the function *approx* is defined by

$$\begin{array}{lll} \mathit{approx} : \mathbb{N} \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] & & \\ \mathit{approx}\ 0 \qquad\quad \_ & = \bot & \\ \mathit{approx}\ (n+1)\ [\,] & = [\,] & \\ \mathit{approx}\ (n+1)\ (x : xs) & = x : \mathit{approx}\ n\ xs. & \end{array} \tag{74}$$

Note that this definition takes place on the meta-level, since the natural numbers $\mathbb{N}$ do not correspond to any type in our language. (Note also that, just as above, we do not write out semantic brackets $[\![\cdot]\!]$ or contexts.)

We have the following:

$$\begin{array}{ll}
 & \forall \text{ total } xs \text{ containing total, finite numbers.} \\
 & \quad (\mathit{diffs} \circ \mathit{sums})\ xs = xs \\
\Leftrightarrow & \{\text{approximation lemma}\} \\
 & \forall \text{ total } xs \text{ containing total, finite numbers.} \\
 & \quad \forall n \in \mathbb{N}.\ \mathit{approx}\ n\ ((\mathit{diffs} \circ \mathit{sums})\ xs) = \mathit{approx}\ n\ xs \\
\Leftrightarrow & \{\text{predicate logic}\} \\
 & \forall n \in \mathbb{N}.\ \forall \text{ total } xs \text{ containing total, finite numbers.} \\
 & \quad \mathit{approx}\ n\ ((\mathit{diffs} \circ \mathit{sums})\ xs) = \mathit{approx}\ n\ xs \\
\Leftrightarrow & \{\text{definition of } \mathit{diffs}, \mathit{sums} \text{ and } \circ\} \\
 & \forall n \in \mathbb{N}.\ \forall \text{ total } xs \text{ containing total, finite numbers.} \\
 & \quad \mathit{approx}\ n\ (\mathit{unfoldr}\ \mathit{step}\ (0, \mathit{unfoldr}\ \mathit{next}\ (0, xs))) = \mathit{approx}\ n\ xs \\
\Leftarrow & \{\text{generalise, 0 is total and finite}\} \\
 & \forall n \in \mathbb{N}.\ \forall \text{ total } xs \text{ containing total, finite numbers.} \\
 & \quad \forall \text{ total and finite } y. \\
 & \qquad \mathit{approx}\ n\ (\mathit{unfoldr}\ \mathit{step}\ (y, \mathit{unfoldr}\ \mathit{next}\ (y, xs))) = \mathit{approx}\ n\ xs.
\end{array}$$

We proceed by induction on the natural number n. The $n = 0$ case is trivial. For $n = k + 1$ we have two cases, $xs = [\,]$ and $xs = z : zs$ (with z

being a total, finite natural number etc.). The first case is easy,

$$\begin{aligned}
& \mathit{approx}\ (k+1)\ (\mathit{unfoldr}\ \mathit{step}\ (y, \mathit{unfoldr}\ \mathit{next}\ (y, [\,]))) \\
= \ & \{\text{definition of } \mathit{unfoldr} \text{ and } \mathit{next}\} \\
& \mathit{approx}\ (k+1)\ (\mathit{unfoldr}\ \mathit{step}\ (y, [\,])) \\
= \ & \{\text{definition of } \mathit{unfoldr} \text{ and } \mathit{step}\} \\
& \mathit{approx}\ (k+1)\ [\,],
\end{aligned}$$

whereas the second one requires a little more work:

$$\begin{aligned}
& \mathit{approx}\ (k+1)\ (\mathit{unfoldr}\ \mathit{step}\ (y, \mathit{unfoldr}\ \mathit{next}\ (y, z : \mathit{zs}))) \\
= \ & \{\text{definition of } \mathit{unfoldr} \text{ and } \mathit{next}\} \\
& \mathit{approx}\ (k+1)\ (\mathit{unfoldr}\ \mathit{step}\ (y, y+z : \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= \ & \{\text{definition of } \mathit{unfoldr} \text{ and } \mathit{step}\} \\
& \mathit{approx}\ (k+1)\ ((y{+}z){-}y : \mathit{unfoldr}\ \mathit{step}\ (y{+}z, \mathit{unfoldr}\ \mathit{next}\ (y{+}z, \mathit{zs}))) \\
= \ & \{(y+z)-y = z \text{ for } z,\, y \text{ total and finite}\} \\
& \mathit{approx}\ (k+1)\ (z : \mathit{unfoldr}\ \mathit{step}\ (y+z, \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= \ & \{\text{definition of } \mathit{approx}\} \\
& z : \mathit{approx}\ k\ (\mathit{unfoldr}\ \mathit{step}\ (y+z, \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= \ & \{\text{inductive hypothesis, } y+z \text{ is total and finite}\} \\
& z : \mathit{approx}\ k\ \mathit{zs} \\
= \ & \{\text{definition of } \mathit{approx}\} \\
& \mathit{approx}\ (k+1)\ (z : \mathit{zs}).
\end{aligned}$$

9.3 Discussion

The last proof above, based on reasoning using domain-theoretic methods, is clearly more concise than the previous one (even if we take into account that we were somewhat more detailed in the first one). It also proves a stronger result since it is not limited to finite lists.

When we compare to the example in Section 2 we see that we were fortunate not to have to explicitly propagate any preconditions through functions in the domain-theoretic proof here. Notice especially the third step in the last case above. The variables y and z were assumed to be finite and total, and hence the lemma $(y + z) - y = z$ could immediately be applied.

There is of course the possibility that the set-theoretic implementation and proof are unnecessarily complex. Note for instance that the domain-theoretic variants work equally well in the set-theoretic world, if we go for coinductive instead of inductive lists, and replace the approximation lemma with the *take* lemma [HG01]. Using such techniques in a sense leads to more robust results, since they never require preconditions of the kind above to be propagated manually.

However, since inductive and coinductive types are not easily mixed we cannot always go this way. If we e.g. want to process the result of *sums* using a fold, we cannot use coinductive lists. In general we cannot use hylomorphisms [MFP91], unfolds followed by folds, in a total setting. If we want or need to use a hylomorphism, then we have to use a partial language.

10 Strict languages

We can treat strict languages using the framework developed so far; at least the somewhat odd language introduced below. For simplicity we reuse the previously given set-theoretic semantics, and also all of the domain-theoretic semantics, except for one rule, the one for application.

More explicitly, we define the domain-theoretic, strict semantics $[\![\cdot]\!]_\perp$ by

$$[\![\sigma]\!]_\perp = [\![\sigma]\!] \tag{75}$$

for all types. For terms we let application be strict,

$$[\![t_1 t_2]\!]_\perp \,\rho = \begin{cases} ([\![t_1]\!]_\perp \,\rho) \mathbin{@} ([\![t_2]\!]_\perp \,\rho)\,, & [\![t_2]\!]_\perp \,\rho \neq \perp, \\ \perp, & \text{otherwise.} \end{cases} \tag{76}$$

Abstractions are treated just as before,

$$[\![\lambda x.t]\!]_\perp \,\rho = \lambda v.\, [\![t]\!]_\perp \,\rho[x \mapsto v]\,, \tag{77}$$

and whenever t is not an application or abstraction we let

$$[\![t]\!]_\perp \,\rho = [\![t]\!]\,\rho. \tag{78}$$

We then define a syntactic translation * on $\mathcal{L}_1$ with the intention of proving that $[\![t]\!]_\perp \,\rho = [\![t^*]\!]\,\rho$. The translation is as follows,

$$t^* = \begin{cases} \mathsf{seq}\ {t_2}^*\ ({t_1}^*\ {t_2}^*), & t = t_1\ t_2, \\ \lambda x.{t_1}^*, & t = \lambda x.t_1, \\ t, & \text{otherwise,} \end{cases} \tag{79}$$

and the desired property follows easily by induction over the structure of terms. It is also easy to prove that $\langle\!\langle t \rangle\!\rangle\,\rho = \langle\!\langle t^* \rangle\!\rangle\,\rho$.

Given these properties we can easily prove the variants of the main theorem (48) and its corollary (49) that result from replacing $[\![\cdot]\!]$ with $[\![\cdot]\!]_\perp$.

11 Related work

The notion of totality used above is very similar to that used by Scott [Sco76]. Aczel's interpretation of Martin-Löf type theory [Acz77] is also based on similar ideas, but types are modelled as predicates instead of PERs. That work has been extended by e.g. Smith [Smi84], who interprets a polymorphic variant of Martin-Löf type theory in an untyped language which shares many properties with our language $\mathcal{L}_2$; he does not consider coinductive types or seq, though. Beeson [Bee82] considers a variant of Martin-Löf type theory with W-types. W-types can be used to model strictly positive inductive and coinductive types [Dyb97, AAG]. Modelling coinductive types can also be done in other ways [Hal87], and the standard trick of coding non-strict evaluation using function spaces (*force* and *delay*) should also be applicable. Furthermore it seems as if Per Martin-Löf, in unpublished work, considered lifted function spaces in a setting similar to [Smi84].

The method we use to relate the various semantic models is basically that of Friedman [Fri75]; his method is more abstract, but defined for a language with only base types, natural numbers and functions.

There is a vast body of text written on the subject of Aczel interpretations, PER models of types, and so on, and some results may be known as folklore without having been published. This text can be seen as a summary of some results, most of them previously known in one form or another, that we consider important for reasoning about functional programs. Furthermore we apply the ideas to the problem of reasoning about programs, instead of using them only to interpret one theory in another. Others have made similar attempts, e.g. [Dyb85].

12 Discussion and future work

We have justified reasoning about functional languages containing partial and infinite values and lifted types, including lifted functions, using total methods.

It should be clear from the examples above that using total methods can sometimes be cheaper than partial ones, and sometimes more expensive. We have not performed any quantitative measurements, so we cannot judge the relative frequency of these two outcomes. One reasonable conclusion is that it would be good if total and partial methods could be mixed without large overheads. We have not experimented with that, but can still make some remarks.

First it should be noted that $\sim$ is not a congruence. We can still use an established fact like $x \sim y$ by translating the statement into a form using preconditions and equality, like we did in Section 8. This translation is easy, but may result in many nontrivial preconditions, perhaps more preconditions than partial reasoning would lead to. When this is not the case it seems as if using total reasoning in some leafs of a proof, and then partial reasoning on the top-level, should work out nicely.

Another observation is that, even if some term t is written in a partial style (using fix), we may still have $[\![t]\!] \in \mathrm{dom}(\sim)$. As an example this would be the case if we implemented *foldr* (see Section 9.1) using fix instead of fold. Hence, if we explicitly prove that $[\![t]\!] \in \mathrm{dom}(\sim)$ we can then use t in a total setting. This proof may be expensive, but enables us to use total reasoning on the top-level, with partial reasoning in some of the leafs.

Now on to other issues. An obvious question is whether one can extend the results to more advanced languages incorporating polymorphism or type constructors. Adding polymorphism would give us an easy way to transform free theorems [Rey83, Wad89] from the set-theoretic side (well, perhaps not set-theoretic [Rey84]) to the domain-theoretic one. It should be interesting to compare those results to other work involving free theorems and seq [JV04].

However, the main motivation for going to a more advanced type system is that we want the results to be applicable to Haskell; matching more features of Haskell's type system is of course important for that goal. Still, the current results should be sufficient to reason about monomorphic Haskell programs. There is one important caveat to this statement, though. Haskell uses the sums-of-products style of data type definitions. When simulating such definitions using binary type constructors extra bottoms are introduced. As an example, $[\![\mu(K_1 + \mathit{Id})]\!]$ contains the different values $\mathit{in}\ \mathit{inl}(\bot)$ and $\mathit{in}\ \mathit{inl}(\star)$; the Haskell data type *Nat* from Section 2 does not contain an analogue of $\mathit{in}\ \mathit{inl}(\bot)$, since the constructor *Zero* is nullary. One solution to this problem is to restrict the types used on the Haskell side to analogues of those available in $\mathcal{L}_1''$. It may also be possible to provide a syntactic translation, using seq, similarly to our treatment of strict languages in Section 10. Finally there is of course the hard way out; reworking the theory using sums-of-products style data types.

Acknowledgements

Thanks to Thierry Coquand and Peter Dybjer for pointing out related work and giving us feedback. Thanks also to Koen Claessen, Jörgen Gustavsson, Patrik Jansson, Ulf Norell, K. V. S. Prasad, David Sands and Josef Svenningsson for participating in discussions and/or giving feedback, and once

more to Patrik for proof-reading various drafts.

References

[AAG] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers – constructing strictly positive types. To appear in Theoretical Computer Science.

[Acz77] Peter Aczel. The strength of Martin-Löf's intuitionistic type theory with one universe. In Seppo Miettinen and Jouko Vänänen, editors, *Proceedings of Symposia in Mathematical Logic, Oulu, 1974, and Helsinki, 1975*, pages 1–32, University of Helsinki, Department of Philosophy, 1977.

[BdBH$^+$91] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.

[BdM96] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[Bee82] M. Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.

[Bir87] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[Dan05] Nils Anders Danielsson. Personal web page, available at `http://www.cs.chalmers.se/~nad/`, 2005.

[Dyb85] Peter Dybjer. Program verification in a logical theory of constructions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 334–349. Springer-Verlag, September 1985. Appears in revised form as Programming Methodology Group Report 26, University of Göteborg and Chalmers University of Technology, June 1986.

[Dyb97] Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176:329–335, 1997.

[FM91] Maarten M Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, January 1991.

[Fri75] Harvey Friedman. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium: Symposium on Logic held at Boston, 1972-73*, number 453 in Lecture Notes in Mathematics, pages 22–37. Springer, 1975.

[Gib96] Jeremy Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, 6(3):535–562, 1996.

[Hal87] Lars Hallnäs. An intensional characterization of the largest bisimulation. *Theoretical Computer Science*, 53(2–3):335–343, 1987.

[HG01] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, August 2001.

[Jeu90] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[JV04] 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.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[Mee86] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[Pri02] Hilary A. Priestley. Ordered sets and complete lattices, a primer for computer science. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 2, pages 21–78. Springer-Verlag, 2002.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, 1983.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *LNCS*, pages 145–156. Springer-Verlag, 1984.

[Sco76] Dana Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, September 1976.

[Smi84] Jan Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.