Fast and Loose Reasoning is Morally Correct *

Nils Anders Danielsson John Hughes
Patrik Jansson
Chalmers University of Technology
{nad,rjmh,patrikj}@cs.chalmers.se

Jeremy Gibbons
Oxford University Computing Laboratory
Jeremy.Gibbons@comlab.ox.ac.uk

Abstract

Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.

Two languages are defined, one total and one partial, with identical syntax. The semantics of the partial language includes partial and infinite values, and all types are lifted, including the function spaces. A partial equivalence relation (PER) is then defined, the domain of which is the total subset of the partial language. For types not containing function spaces the PER relates equal values, and functions are related if they map related values to related values.

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. It is also shown that the PER gives rise to a bicartesian closed category which can be used to reason about values in the domain of the relation.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Languages, theory, verification

Keywords Equational reasoning, partial and total languages, non-strict and strict languages, partial and infinite values, lifted types, inductive and coinductive types

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

$$curry \circ uncurry = id, \tag{1}$$

$$(fst\ x, snd\ x) = x, \text{ and} \tag{2}$$

$$fst\ (x, y) = x, \tag{3}$$

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, the Squiggolers [BdM96, Jeu90, BdBH+91, MFP91]. There is just one problem. In current real programming languages such as Haskell [PJ03] and ML [MTHM97], they are not generally valid.

The reason these laws fail is the presence of the undefined value $\bot$, and the fact that, in Haskell, $\bot$, $\lambda x.\bot$ and $(\bot, \bot)$ are all different (violating the first two laws above), while in ML, $\bot$, $(x, \bot)$ and $(\bot, y)$ 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 $(x, \bot) = (y, \bot)$ 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 $\bot$-cases much of the time.

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

* 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' funds.

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. Section 8 provides a more abstract result, showing how the PER gives rise to a category with many nice properties which can be used to reason about programs. We go back to our earlier example and show how it fits in with the theory in Section 9. We also exhibit another example where reasoning directly about the domain-theoretic semantics of a program may be preferable (Section 10). Section 11 recasts the theory for a strict language, Section 12 discusses related work, and Section 13 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 are available from Danielsson's web page [Dan05].

2. Propagating preconditions

Let us begin with an example. Say that we need to prove that the function $map\ (\lambda x.y + x) \circ reverse :: [Nat] \to [Nat]$ has a left inverse $reverse \circ map\ (\lambda x.x - y)$. (All code in this section uses Haskell-like syntax.) In a total language we would do it more or less like this:

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

Note the lemmas used for the proof, especially

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

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

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,

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

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

$$(+) = fold\ Succ, \text{ and} \tag{7}$$

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

where $fold :: (a \to a) \to a \to Nat \to a$ is the fold over natural numbers ($fold\ s\ z\ n$ replaces all occurrences of $Succ$ in n with s, and $Zero$ with z), and $pred :: Nat \to Nat$ is the predecessor function with $pred\ Zero = Zero$. The other functions and types are all standard [PJ03]; this implies 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, since

$$(Succ\ Zero + Succ\ \bot) - Succ\ Zero = \bot \neq Succ\ \bot \text{ and} \tag{9}$$

$$reverse\ (reverse\ (Zero : \bot)) = \bot \neq Zero : \bot, \tag{10}$$

and infinite values, since

$$(fix\ Succ + Zero) - fix\ Succ = \bot \neq Zero \text{ and} \tag{11}$$

$$reverse\ (reverse\ (repeat\ Zero)) = \bot \neq repeat\ Zero. \tag{12}$$

(Here fix is the fixpoint combinator, i.e. $fix\ Succ$ is the "infinite" lazy natural number. The application $repeat\ x$ yields an infinite list containing only x.) Note that $id \circ f = f$ also fails, since we have lifted function spaces and $id \circ \bot = \lambda x.\bot \neq \bot$, but that does not affect this example since $reverse \neq \bot$.

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 stay in the partial language but restrict our inputs to total, finite values.

Let us see what the proof looks like then. We have to η-expand our property, and assume that $xs :: [Nat]$ is a total, finite list and that $y :: Nat$ 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 values. The concepts of totality and finiteness are discussed in more detail in Sections 5 and 9, respectively.) We get

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

Comparing to the previous proof we see that all steps are more or less identical, using similar lemmas, except for the second step, where two new lemmas are required. How did that step become so unwieldy? The problem is that, although we know that $(\lambda x.x-y) \circ (\lambda x.y+x) = id$ given total input, and also that xs only contains total natural numbers, we have to manually *propagate* this precondition through $reverse$ and map.

One view of 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.

On the other hand, 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 total, 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 yes, with a little extra effort, and proving this is what many of the sections below will be devoted to. In Section 9 we come back to this example and spell out in full detail what "a little extra effort" boils down to in this case.

3. Language

This section defines 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 13.

3.1 Static semantics

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

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

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

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

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 types μF and νF are inductive and coinductive types, respectively. As an example, in the set-theoretic semantics introduced below $\mu(K_1 + Id)$ represents finite natural numbers, and $\nu(K_1 + Id)$ represents 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

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

However, whenever fix is not explicitly mentioned, the language discussed is $\mathcal{L}_1$ (or the restriction $\mathcal{L}_1'$ 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.

3.2 Dynamic semantics

Before we define the semantics of the languages we need to introduce some notation. We will use both sets and

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

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

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

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.

pointed ω-complete partial orders (CPOs). For CPOs $\cdot_\perp$ is the lifting operator, and $\langle \cdot \to \cdot \rangle$ is the continuous function space constructor. Furthermore, for both sets and CPOs, $\times$ 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 set/CPO is denoted by 1, with $\star$ as the only element. The constructor for the (set- or domain-theoretic) semantic domain of the recursive type T, where T is μF or νF, is denoted by in_T, and the corresponding destructor is denoted by out_T (often with omitted indices). For more details about in and out, see below. Since we use lifted function spaces, we use special notation for lifted function application,

$$f @ x = \begin{cases} \perp, & f = \perp, \\ f\ x, & \text{otherwise.} \end{cases} \tag{17}$$

(This operator is left associative with the same precedence as ordinary function application.) 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, 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.

Now, two different denotational semantics are defined for the languages introduced above, one domain-theoretic ($[\![\cdot]\!]$) and one set-theoretic ($\langle\!\langle\cdot\rangle\!\rangle$). (Note that when t is closed we sometimes use $[\![t]\!]$ as a shorthand for $[\![t]\!]\,\rho$, and similarly for $\langle\!\langle\cdot\rangle\!\rangle$.) The domain-theoretic semantics is modelled on languages like Haskell and can handle general recursion. The set-theoretic semantics is modelled on total languages and is only defined for terms in $\mathcal{L}_1$. In Section 7 we will show how results obtained using the set-theoretic semantics can be transformed into results on the domain-theoretic side.

The semantic domains for all 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 $in_{\mu F} \in \langle\!\langle F\ \mu F \to \mu F \rangle\!\rangle$. The inverse of $in_{\mu F}$ exists and, as noted above, is denoted by $out_{\mu F}$. Initiality of $in_{\mu F}$ implies that for any function $f \in \langle\!\langle F\ \sigma \to \sigma \rangle\!\rangle$ there is a unique function $fold_F\ f \in \langle\!\langle \mu F \to \sigma \rangle\!\rangle$ satisfying the universal property

$$\forall h \in \langle\!\langle \mu F \to \sigma \rangle\!\rangle .\ h = fold_F\ f \ \Leftrightarrow\ h \circ in_{\mu F} = f \circ F\ h. \quad (18)$$

This is how $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$ is defined. To define $\langle\!\langle \mathsf{unfold}_F \rangle\!\rangle$ we go via the *final* object $out_{\nu F}$ 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 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, which is defined in Figure 5.

If we were to define $[\![\mathsf{fold}_F]\!]$ using the same method as for $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$, 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 folds if all functions involved are strict [FM91]. For symmetry we also define $[\![\mathsf{unfold}_F]\!]$ using an explicit fixpoint; that does not affect its universality property.

The semantics of fix is, as usual, given by a least fixpoint construction.

We have been a little sloppy above, in that 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.

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

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

(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, including all those in this paper. For example, given the PER defined in Section 5, we have that $[inl(\{c\})]$ denotes the same equivalence class no matter which element in c is chosen.

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

$$[\![\mu F]\!] = \left\{ \begin{array}{l} \text{The codomain of the initial object in} \\ L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\perp). \end{array} \right.$$

$$\langle\!\langle \mu F \rangle\!\rangle = \left\{ \begin{array}{l} \text{The codomain of the initial object in} \\ F\text{-}\mathsf{Alg}(\mathsf{SET}). \end{array} \right.$$

$$[\![\nu F]\!] = \left\{ \begin{array}{l} \text{The domain of the final object in} \\ L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO}). \end{array} \right.$$

$$\langle\!\langle \nu F \rangle\!\rangle = \{\, \text{The domain of the final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET}).$$

Figure 4: Semantic domains for types.

5. Moral equality

We will now inductively define a family of PERs $\sim_\sigma$ on the domain-theoretic semantic domains; with $Rel(\sigma) = \wp([\![\sigma]\!]^2)$ we will have $\sim_\sigma \in Rel(\sigma)$. (Here $\wp(X)$ is the power set of X. The index σ will sometimes be omitted.)

If two values are related by $\sim$, then we say that they are *morally equal*. We use moral equality to formalise totality: a value $x \in [\![\sigma]\!]$ is said to be *total* iff $x \in \mathrm{dom}(\sim_\sigma)$. The intention is that if σ does not contain function spaces, then we should have $x \sim_\sigma y$ iff x and y are equal, total values. For functions we will have $f \sim g$ iff f and g map (total) related values to related values.

The definition of totality given here should correspond to basic intuition. Sometimes another definition is used instead, where $f \in [\![\sigma \to \tau]\!]$ is total iff $f @ x = \perp$ implies that $x = \perp$. That definition is not suitable for non-strict languages where most semantic domains are not flat. As a simple example, consider $[\![\mathsf{fst}]\!]$; we will have $[\![\mathsf{fst}]\!] \in \mathrm{dom}(\sim)$, so $[\![\mathsf{fst}]\!]$ is total according to our definition, but $[\![\mathsf{fst}]\!] @ (\perp, \perp) = \perp$.

Given the family of PERs $\sim$ 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 $\sim_{\sigma\to\tau}$ is a logical relation, i.e. we have the following definition for function spaces:

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

We need to ensure explicitly that f and g are non-bottom because some of the PERs will turn out to have $\perp \in \mathrm{dom}(\sim)$ or $\mathrm{dom}(\sim) = \emptyset$.

Pairs are related if corresponding components are related:

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

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

Figure 5: Lifting of functors.

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

$$
[\![\mathsf{case}]\!] = \lambda v\,f_1\,f_2.\begin{cases}\bot, & v = \bot \\ f_1@v_1, & v = inl(v_1) \\ f_2@v_2, & v = inr(v_2)\end{cases}
$$

$$
\langle\!\langle \mathsf{case} \rangle\!\rangle = \lambda v\,f_1\,f_2.\begin{cases}f_1\ v_1, & v = inl(v_1) \\ f_2\ v_2, & v = inr(v_2)\end{cases}
$$

$$
[\![\mathsf{in}_{\mu F}]\!] = \begin{cases}\text{The initial object in } L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\bot), \\ \text{viewed as a morphism in } \mathsf{CPO}.\end{cases}
$$

$$
\langle\!\langle \mathsf{in}_{\mu F} \rangle\!\rangle = \begin{cases}\text{The initial object in } F\text{-}\mathsf{Alg}(\mathsf{SET}), \text{ viewed as} \\ \text{a morphism in } \mathsf{SET}.\end{cases}
$$

$$
[\![\mathsf{out}_{\nu F}]\!] = \begin{cases}\text{The final object in } L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO}), \\ \text{viewed as a morphism in } \mathsf{CPO}.\end{cases}
$$

$$
\langle\!\langle \mathsf{out}_{\nu F} \rangle\!\rangle = \begin{cases}\text{The final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET}), \text{ viewed} \\ \text{as a morphism in } \mathsf{SET}.\end{cases}
$$

$$
[\![\mathsf{fold}_F]\!] = \lambda f.\,[\![\mathsf{fix}]\!]@(\lambda g.f \circ [\![F]\!]@g \circ [\![\mathsf{out}_{\mu F}]\!])
$$

$$
\langle\!\langle \mathsf{fold}_F \rangle\!\rangle = \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Alg}(\mathsf{SET}) \\ \text{from } \langle\!\langle \mathsf{in}_{\mu F} \rangle\!\rangle \text{ to } f, \text{ viewed as a morphism} \\ \text{in } \mathsf{SET}.\end{cases}
$$

$$
[\![\mathsf{unfold}_F]\!] = \lambda f.\,[\![\mathsf{fix}]\!]@(\lambda g.\,[\![\mathsf{in}_{\nu F}]\!] \circ [\![F]\!]@g \circ f)
$$

$$
\langle\!\langle \mathsf{unfold}_F \rangle\!\rangle = \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Coalg}(\mathsf{SET}) \\ \text{from } f \text{ to } \langle\!\langle \mathsf{out}_{\nu F} \rangle\!\rangle, \text{ viewed as a} \\ \text{morphism in } \mathsf{SET}.\end{cases}
$$

Figure 6: Semantics of well-typed terms, for some context ρ mapping variables to semantic values. The semantics of $\mathsf{in}_{\nu F}$ and $\mathsf{out}_{\mu F}$ are the inverses of the semantics of $\mathsf{out}_{\nu F}$ and $\mathsf{in}_{\mu F}$, respectively.

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

$$
\begin{aligned}
&x \sim_{\sigma+\tau} y \ \Leftrightarrow \\
&\quad (\exists x_1, y_1 \in [\![\sigma]\!]\,. \\
&\qquad\quad x = inl(x_1) \ \wedge\ y = inl(y_1) \ \wedge\ x_1 \sim_\sigma y_1) \ \vee \\
&\quad (\exists x_2, y_2 \in [\![\tau]\!]\,. \\
&\qquad\quad x = inr(x_2) \ \wedge\ y = inr(y_2) \ \wedge\ x_2 \sim_\tau y_2)\,.
\end{aligned} \tag{22}
$$

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 family of PERs.

5.2 Recursive types

The definition for recursive types is trickier. Consider lists. When should one 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

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

We formalise the intuition embodied in (24) by defining a relation transformer $RT(F)$ for each functor F,

$$
\begin{aligned}
&RT(F) \in Rel(\mu F) \to Rel(\mu F) \\
&RT(F)(X) = \\
&\qquad \big\{\ (in\ x, in\ y)\ \big|\ (x, y) \in RT'_{\mu F}(F)(X)\ \big\}\,.
\end{aligned} \tag{25}
$$

The helper $RT'_\sigma(F)$ is defined by

$$
\begin{aligned}
&RT'_\sigma(F) \in Rel(\sigma) \to Rel(F\ \sigma) \\
&RT'_\sigma(Id)(X) \quad\ = X \\
&RT'_\sigma(K_\tau)(X) \quad\ = \sim_\tau \\
&RT'_\sigma(F_1 \times F_2)(X) = \\
&\qquad \left\{\ ((x_1, x_2), (y_1, y_2))\ \middle|\ \begin{matrix}(x_1, y_1) \in RT'_\sigma(F_1)(X), \\ (x_2, y_2) \in RT'_\sigma(F_2)(X)\end{matrix}\ \right\} \\
&RT'_\sigma(F_1 + F_2)(X) = \\
&\quad \big\{\ (inl(x_1)\,, inl(y_1))\ \big|\ (x_1, y_1) \in RT'_\sigma(F_1)(X)\ \big\} \cup \\
&\quad \big\{\ (inr(x_2)\,, inr(y_2))\ \big|\ (x_2, y_2) \in RT'_\sigma(F_2)(X)\ \big\}\,.
\end{aligned} \tag{26}
$$

The relation transformer $RT(F)$ is defined for inductive types. However, replacing μF with νF in the definition is enough to yield a transformer suitable for coinductive types.

Now, note that $RT(F)$ is a monotone operator on the complete lattice $(Rel(\mu F)\,, \subseteq)$. This implies that it has both least and greatest fixpoints [Pri02], which leads to the following definitions:

$$
x \sim_{\mu F} y \ \Leftrightarrow\ (x, y) \in \mu RT(F) \text{ and} \tag{27}
$$

$$
x \sim_{\nu F} y \ \Leftrightarrow\ (x, y) \in \nu RT(F)\,. \tag{28}
$$

These definitions may not be entirely transparent. If we go back to the list example and expand the definition of

$RT(K_1 + (K_\sigma \times Id))$ we get

$$\begin{array}{l} RT(K_1 + (K_\sigma \times Id))(X) = \\ \quad \{\, (in\ (inl(\star)), in\ (inl(\star)))\,\} \cup \\ \quad \left\{ \begin{array}{l} (in\ (inr((x, xs))), \\ \ in\ (inr((y, ys)))) \end{array} \;\middle|\; \begin{array}{l} x, y \in [\![\sigma]\!]\,,\ x \sim y, \\ (xs, ys) \in X \end{array} \right\}. \end{array} \tag{29}$$

The least and greatest fixpoints of this operator correspond to our original aims for $\sim_{\mu(K_1+(K_\sigma\times Id))}$ and $\sim_{\nu(K_1+(K_\sigma\times Id))}$. (Note that we never consider an infinite inductive list as being total.)

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

$$\forall X \subseteq [\![\mu F]\!]^2\,.\ RT(F)(X) \subseteq X \ \Rightarrow\ \mu RT(F) \subseteq X, \tag{30}$$

and coinduction,

$$\forall X \subseteq [\![\nu F]\!]^2\,.\ X \subseteq RT(F)(X) \ \Rightarrow\ X \subseteq \nu RT(F)\,. \tag{31}$$

Many proofs needed for this paper proceed according to a scheme similar to the following one, 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 it is proved that $\sim$ is a family of PERs [Dan05].

5.3 Properties

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

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

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 is 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 $x = y$.

Continuing with order related properties, it is proved using induction over the type structure that $\sim_\sigma$ is monotone when seen as a function $\sim_\sigma \in [\![\sigma]\!]^2 \to 1_\perp$. This implies that all equivalence classes are upwards closed. We also have (by induction over the type structure) that $\perp \notin \mathrm{dom}(\sim_\sigma)$ for almost all types σ. The only exceptions are given by the grammar

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

Note that $[\![\chi]\!] = \{\perp\}$ for all these types.

The (near-complete) absence of bottoms in $\mathrm{dom}(\sim)$ gives us an easy way to show that related values are not always equal: at most types $[\![\mathsf{seq}]\!] \sim [\![\lambda x.\lambda y.y]\!]$ but $[\![\mathsf{seq}]\!] \neq [\![\lambda x.\lambda y.y]\!]$. This example breaks down when seq is used at type $\chi \to \sigma \to \sigma$ (unless $\mathrm{dom}(\sim_\sigma) = \emptyset$). To be able to prove the fundamental theorem below, let $\mathcal{L}_1'$ denote the language consisting of all terms from $\mathcal{L}_1$ which contain no uses of seq at type $\chi \to \sigma \to \sigma$.

Now, by using induction over the term structure instead of the type structure, and then following the rest of IIICI, it is shown 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 $\mathcal{L}_1'$. In other words, all closed terms in $\mathcal{L}_1'$ denote total values. Note, however, that $[\![\mathsf{fix}]\!] \notin \mathrm{dom}(\sim)$ (at most types) since $[\![\lambda x.x]\!] \in \mathrm{dom}(\sim)$ and $[\![\mathsf{fix}\ (\lambda x.x)]\!] = \perp$.

5.4 Examples

With moral equality defined we can prove a number of laws for $\sim$ which are not true for $=$. As an example, consider η-equality (combined with extensionality):

$$\begin{array}{l} \forall f, g \in [\![\sigma \to \tau]\!]\,. \\ \qquad (\forall x \in [\![\sigma]\!]\,.\ f@x = g@x) \ \Leftrightarrow\ f = g. \end{array} \tag{36}$$

This law is not valid, since the left hand side is satisfied by the distinct values $f = \perp$ and $g = \lambda v.\perp$. On the other hand, the following variant follows immediately from the definition of $\sim$:

$$\begin{array}{l} \forall f, g \in \mathrm{dom}(\sim_{\sigma\to\tau})\,. \\ \qquad (\forall x \in \mathrm{dom}(\sim_\sigma)\,.\ f@x \sim g@x) \ \Leftrightarrow\ f \sim g. \end{array} \tag{37}$$

As another example, consider currying (1). The corresponding statement, $[\![curry \circ uncurry]\!] \sim [\![id]\!]$, is easily proved using the fundamental theorem (35) and the η-law (37) above. We can also prove surjective pairing. Since $p \in \mathrm{dom}(\sim_{\sigma\times\tau})$ implies that $p = (x, y)$ for some $x \in \mathrm{dom}(\sim_\sigma)$ and $y \in \mathrm{dom}(\sim_\tau)$ we get $[\![(\mathsf{fst}\ t, \mathsf{snd}\ t)]\!]\,\rho = [\![t]\!]\,\rho$, given that $[\![t]\!]\,\rho \in \mathrm{dom}(\sim)$.

6. Partial surjective homomorphism

For the main theorem (Section 7) we need to relate values in $\langle\!\langle\sigma\rangle\!\rangle$ to values in $[\sim_\sigma]$, the set of equivalence classes of $\sim_\sigma$. Due to cardinality issues there is in general no total bijection between these sets; consider $\sigma = (Nat \to Nat) \to Nat$ with $Nat = \mu(K_1 + Id)$, for instance. We can define a *partial surjective homomorphism* [Fri75] from $\langle\!\langle\sigma\rangle\!\rangle$ to $[\sim_\sigma]$, though. This means that for each type σ there is a partial, surjective function $j_\sigma \in \langle\!\langle\sigma\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_\sigma]$, which for function types satisfies

$$(j_{\tau_1\to\tau_2}\ f)\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x) \tag{38}$$

whenever $f \in \mathrm{dom}(j_{\tau_1\to\tau_2})$ and $x \in \mathrm{dom}(j_{\tau_1})$. (Here $\mathrel{\tilde{\to}}$ is the partial function space constructor and $\mathrm{dom}(f)$ denotes the domain of the partial function f. Furthermore we define $[f]\ [x] = [f@x]$, which is well-defined.)

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

$$\begin{array}{l} j_{\sigma\times\tau} \in \langle\!\langle\sigma \times \tau\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\sigma\times\tau}] \\ j_{\sigma\times\tau}\ (x, y) = [(\{j_\sigma\ x\}, \{j_\tau\ y\})]\,, \end{array} \tag{39}$$

$$\begin{array}{l} j_{\sigma+\tau} \in \langle\!\langle\sigma + \tau\rangle\!\rangle \mathrel{\tilde{\to}} [\sim_{\sigma+\tau}] \\ j_{\sigma+\tau}\ inl(x) = [inl(\{j_\sigma\ x\})] \\ j_{\sigma+\tau}\ inr(y) = [inr(\{j_\tau\ y\})]\,, \end{array} \tag{40}$$

and

$$\begin{array}{l} j_1 \in \langle\!\langle 1 \rangle\!\rangle \xrightarrow{\sim} [\sim_1] \\ j_1 \star = [\star]\,. \end{array} \tag{41}$$

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

It turns out to be impossible in general to come up with a total definition of j for function spaces. Consider the function $\mathit{isInfinite} \in \langle\!\langle \mathit{CoNat} \to \mathit{Bool} \rangle\!\rangle$ (with $\mathit{CoNat} = \nu(K_1 + \mathit{Id})$ and $\mathit{Bool} = 1 + 1$) given by

$$\mathit{isInfinite}\ n = \begin{cases} \mathit{True}, & j\ n = [\omega]\,, \\ \mathit{False}, & \text{otherwise.} \end{cases} \tag{42}$$

(Here $\omega = [\![\mathsf{unfold}_{K_1+\mathit{Id}}\ \mathsf{inr}\ \star]\!]$ is the infinite "natural number", $\mathit{True} = \mathit{inl}(\star)$ and $\mathit{False} = \mathit{inr}(\star)$.) Any surjective homomorphism j must be undefined for *isInfinite*.

Instead we settle for a partial definition. We employ a technique, originating from Friedman [Fri75], which makes it easy to prove that j is homomorphic: if possible, let $j_{\tau_1 \to \tau_2}\ f$ be the element $g \in [\sim_{\tau_1 \to \tau_2}]$ satisfying

$$\forall x \in \mathrm{dom}(j_{\tau_1})\,.\ g\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x). \tag{43}$$

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 use a lemma stating that $\langle\!\langle \sigma \rangle\!\rangle$ is empty iff $[\sim_\sigma]$ is.

The definition of j for inductive and coinductive types follows the idea outlined for the basic cases above, but is more involved, and we omit it here due to space constraints.

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

$$\rho(x) \in \mathrm{dom}(\sim)\ \wedge\ j\ \rho'(x) = [\rho(x)] \tag{44}$$

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

$$j\ \left(\langle\!\langle t \rangle\!\rangle\ \ \rho'\right) = [[\![t]\!]\ \ \rho]\,. \tag{45}$$

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. Note that the proof depends on the particular definition of j given in Section 6; if we wanted to use a different partial surjective homomorphism then the proof would need to be modified.

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 $\rho_1,\ \rho_1'$ and $\rho_2,\ \rho_2'$ both satisfying the conditions of (44) (for t_1 and t_2, respectively), that

$$\langle\!\langle t_1 \rangle\!\rangle\ \rho_1' = \langle\!\langle t_2 \rangle\!\rangle\ \rho_2' \ \Rightarrow\ [\![t_1]\!]\ \rho_1 \sim [\![t_2]\!]\ \rho_2. \tag{46}$$

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 no longer loose.

If j had been injective, then (46) would have been an equivalence. That would mean that we could handle unequalities ($\neq$). The particular j defined here is not injective, which can be shown using the function $\mathit{isIsInfinite} \in \langle\!\langle (\mathit{CoNat} \to \mathit{Bool}) \to \mathit{Bool} \rangle\!\rangle$ given by

$$\mathit{isIsInfinite}\ f = \begin{cases} \mathit{True}, & f = \mathit{isInfinite}, \\ \mathit{False}, & \text{otherwise.} \end{cases} \tag{47}$$

(*CoNat*, *isInfinite* etc. are defined in Section 6.) We get $j\ \mathit{isIsInfinite} = j\ (\lambda f.\mathit{False})$ (both defined), so j is not injective. In fact, no j which satisfies the main theorem (45) and uses the definition above for function spaces (43) can be injective.

8. Category-theoretic approach

Equation (46) above is useful partly because SET is a well-understood category. For those who prefer to work abstractly instead of working in SET, the following result may be a useful substitute. We define the category $\mathsf{PER}_\sim$ as follows:

Objects The objects are types σ (without any restrictions).

Morphisms The morphisms of type $\sigma \to \tau$ are the elements of $[\sim_{\sigma \to \tau}]$, i.e. equivalence classes of total functions.

Composition $[f] \circ [g] = [\lambda v.f@(g@v)]$.

This category is bicartesian closed, with initial algebras and final coalgebras for (at least) polynomial functors. All the laws that follow from this statement can be used to reason about programs. For instance, it should not be hard to repeat the total proofs from this paper using such laws.

For this method to be immediately useful, the various constructions involved should correspond closely to those in the underlying language. And they do:

Initial object The initial object is $\mu\mathit{Id}$, with the unique morphism of type $\mu\mathit{Id} \to \sigma$ given by $[\lambda v.\bot]$.

Final object The final object is 1, with the unique morphism of type $\sigma \to 1$ given by $[\lambda v.\star]$. Note that $\nu\mathit{Id}$ is isomorphic to 1.

Products The product of σ and τ is $\sigma \times \tau$. The projections are $[[\![\mathsf{fst}]\!]]$ and $[[\![\mathsf{snd}]\!]]$, and given $[f] : \gamma \to \sigma$ and $[g] : \gamma \to \tau$ the unique morphism which "makes the diagram commute" is $[\lambda v.(f@v, g@v)]$.

Coproducts The coproduct of σ and τ is $\sigma + \tau$. The injections are $[[\![\mathsf{inl}]\!]]$ and $[[\![\mathsf{inr}]\!]]$, and given $[f] : \sigma \to \gamma$ and $[g] : \tau \to \gamma$ the unique morphism which "makes the diagram commute" is $[\lambda v.\,[\![\mathsf{case}]\!]@v@f@g]$.

Exponentials The exponential of τ and σ is $\sigma \to \tau$. The apply morphism is $[\lambda(f, x).f@x]$, and currying is given by the morphism $[\lambda f\,x\,y.f@(x, y)]$.

Initial algebras For a polynomial functor F the corresponding initial F-algebra is $(\mu F, [[\![\mathsf{in}_{\mu F}]]\!])$. Given the F-algebra $[f] : F\ \sigma \to \sigma$, the unique homomorphism from $[[\![\mathsf{in}_{\mu F}]]\!]$ is $[[\![\mathsf{fold}_F]\!]@f]$.

Final coalgebras For a polynomial functor F the corresponding final F-coalgebra is $(\nu F, [[\![\mathsf{out}_{\nu F}]]\!])$. Given the F-coalgebra $[f] : \sigma \to F\ \sigma$, the unique homomorphism to $[[\![\mathsf{out}_{\nu F}]]\!]$ is $[[\![\mathsf{unfold}_F]\!]@f]$.

The proofs of these properties are rather easy, and do not require constructions like j.

The partial surjective homomorphism j fits into the category-theoretic picture anyway: it can be extended to a partial functor to $\mathsf{PER}_\sim$ from the category which has types σ as objects, total functions between the corresponding set-theoretic domains $\langle\!\langle \sigma \rangle\!\rangle$ as morphisms, and ordinary function composition as composition of morphisms. The object part of this functor is the identity, and the morphism part is given by the function space case of j.

9. 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 type of the functions becomes $\mathit{ListNat} \to \mathit{ListNat}$, where $\mathit{ListNat}$ is the inductive type $\mu(K_1 + (K_{Nat} \times \mathit{Id}))$ of lists of natural numbers, and Nat is the inductive type $\mu(K_1 + \mathit{Id})$. Note also that the functions $\mathit{reverse}$, map, $(+)$ and $(-)$ can be expressed using folds, so the terms belong to $\mathcal{L}_1$. Finally note that seq is not used at a type $\chi \to \sigma \to \sigma$ with $\bot \in \mathrm{dom}(\sim_\chi)$, so the terms belong to $\mathcal{L}_1'$, and we can make full use of the theory.

Our earlier proof in effect showed that

$$\langle\!\langle \mathit{revMap} \circ \mathit{mapRev} \rangle\!\rangle\ [y \mapsto n] = \langle\!\langle \mathit{id} \rangle\!\rangle \tag{48}$$

for an arbitrary $n \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$, which by (46) implies that

$$[\![\mathit{revMap} \circ \mathit{mapRev}]\!]\ \left[y \mapsto n'\right] \sim_{\mathit{ListNat} \to \mathit{ListNat}} [\![\mathit{id}]\!] \tag{49}$$

whenever $n' \in \mathrm{dom}(\sim_{\mathit{Nat}})$ and $[n'] = j\ n$ for some $n \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$. By the fundamental theorem (35) and the main theorem (45) we have that $[\![t]\!]$ satisfies the conditions for n' 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 (49) by f. The equation implies that $f@\mathit{xs} \sim \mathit{ys}$ whenever $\mathit{xs} \sim \mathit{ys}$. 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 $f@\mathit{xs} = \mathit{xs}$ whenever $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$.

We want $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ to mean the same as "xs is total and finite". We defined totality to mean "related according to $\sim$" in Section 5, so $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ iff 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 $\mathit{size}_\sigma \in \mathrm{dom}(\sim_\sigma) \to \mathbb{N}$ which has the property that $\mathit{size}\ x' < \mathit{size}\ x$ whenever x' is a structurally smaller part of x, such as with $x = \mathit{inl}(x')$ or $x = \mathit{in}\ (x', x'')$.

Hence we have arrived at the statement proved by the more elaborate proof in Section 2: for all total and finite lists xs and all total and finite natural numbers n,

$$[\![(\mathit{revMap} \circ \mathit{mapRev})\ \mathit{xs}]\!]\ [y \mapsto n] = [\![\mathit{xs}]\!]\,. \tag{50}$$

This means that we have proved a result about the partial language without having to manually propagate preconditions. At first glance it may seem as if the auxiliary arguments spelled out in this section make using total methods more expensive than we first indicated. However, note that the various parts of these arguments only need to be carried out at most once for each type. They do not need to be repeated for every new proof.

10. 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 $\mathit{ListNat} \to \mathit{ListNat}$, with the inductive types $\mathit{ListNat}$ and Nat defined just like in Section 9. The function sums takes a list of numbers and calculates their running sum, and diffs performs the left inverse operation, along the lines of

$$\mathit{sums}\ [3,1,4,1,5] = [3,4,8,9,14], \text{ and} \tag{51}$$

$$\mathit{diffs}\ [3,4,8,9,14] = [3,1,4,1,5] \tag{52}$$

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

$$\langle\!\langle \mathit{diffs} \circ \mathit{sums} \rangle\!\rangle = \langle\!\langle \mathit{id} \rangle\!\rangle\,. \tag{53}$$

We do that in Section 10.1. Alternatively, we can implement the functions in $\mathcal{L}_2$ and prove that

$$[\![\mathit{diffs} \circ \mathit{sums}]\!]@\mathit{xs} = \mathit{xs} \tag{54}$$

for all total, finite lists $\mathit{xs} \in [\![\mathit{ListNat}]\!]$ containing total, finite natural numbers. That is done in Section 10.2. We then compare the experiences in Section 10.3.

10.1 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:

$$\begin{aligned}
&\mathit{foldr} : (\sigma \to (\tau \to \tau)) \to \tau \\
&\qquad \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \to \tau \\
&\mathit{foldr}\ f\ x = \\
&\quad \mathsf{fold}_{K_1 + (K_\sigma \times \mathit{Id})}\ (\lambda y.\mathsf{case}\ y\ (\lambda_.x) \\
&\qquad\qquad (\lambda p.f\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p))).
\end{aligned} \tag{55}$$

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

$$\begin{aligned}
&\mathit{sums} : \mathit{ListNat} \to \mathit{ListNat} \\
&\mathit{sums} = \mathit{foldr}\ \mathit{add}\ [\,],
\end{aligned} \tag{56}$$

where

$$\begin{aligned}
&\mathit{add} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat} \\
&\mathit{add}\ x\ \mathit{ys} = x : \mathit{map}\ (\lambda y.x + y)\ \mathit{ys}.
\end{aligned} \tag{57}$$

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:

$$\begin{aligned}
&\mathit{map} : (\sigma \to \tau) \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \\
&\qquad \to \mu(K_1 + (K_\tau \times \mathit{Id})) \\
&\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ \mathit{ys}.f\ x : \mathit{ys})\ [\,].
\end{aligned} \tag{58}$$

The definition of diffs uses similar techniques:

$$\begin{aligned}
&\mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat} \\
&\mathit{diffs} = \mathit{foldr}\ \mathit{sub}\ [\,],
\end{aligned} \tag{59}$$

where

$$\begin{aligned}
&\mathit{sub} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat} \\
&\mathit{sub}\ x\ \mathit{ys} = x : \mathit{toHead}\ (\lambda y.y - x)\ \mathit{ys}.
\end{aligned} \tag{60}$$

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

$$\begin{aligned}
&\mathit{toHead} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{ListNat} \to \mathit{ListNat} \\
&\mathit{toHead}\ f\ (y : \mathit{ys}) = f\ y : \mathit{ys} \\
&\mathit{toHead}\ f\ [\,] \qquad = [\,].
\end{aligned} \tag{61}$$

Now let us prove (53). We can use fold fusion [BdM96],

$$\begin{aligned}
&g \circ \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e' \\
&\quad \Leftarrow\ g\ e = e'\ \wedge\ \forall x, y.\ g\ (f\ x\ y) = f'\ x\ (g\ y).
\end{aligned} \tag{62}$$

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

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

The first conjunct is trivial, and the second one can be proved by using the lemmas

$$(\lambda y.y - x) \circ (\lambda y.x + y) = \mathit{id} \tag{63}$$

and

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

To prove the second lemma we use fold-map fusion [BdM96],

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

We have

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

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

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

and

$$\begin{array}{l} \forall y, \mathit{ys}.\ \ \mathit{toHead}\ (\lambda y.x + y)\ (\mathit{sub}\ y\ \mathit{ys}) = \\ \qquad (\mathit{sub} \circ (\lambda y.x + y))\ y\ (\mathit{toHead}\ (\lambda y.x + y)\ \mathit{ys}). \end{array} \tag{67}$$

The first one is yet again trivial, and the second one can be proved by using the lemma

$$\lambda z.z - y = (\lambda z.z - (x + y)) \circ (\lambda y.x + y). \tag{68}$$

10.2 Using partial reasoning

Let us now see what we can accomplish when we are not restricted to 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]:

$$\begin{array}{l} \mathit{unfoldr} : (\tau \to (1 + (\sigma \times \tau))) \to \tau \\ \qquad\qquad \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \\ \mathit{unfoldr}\ f\ b = \mathsf{case}\ (f\ b) \\ \qquad\qquad\qquad\qquad (\lambda\_.[\,]) \\ \qquad\qquad\qquad\qquad (\lambda p.\mathsf{fst}\ p : \mathit{unfoldr}\ f\ (\mathsf{snd}\ p)). \end{array} \tag{69}$$

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*,

$$\begin{array}{l} \mathit{sums} : \mathit{ListNat} \to \mathit{ListNat} \\ \mathit{sums}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{next}\ (0, \mathit{xs}), \end{array} \tag{70}$$

with helper *next*,

$$\begin{array}{l} \mathit{next} : (\mathit{Nat} \times \mathit{ListNat}) \\ \qquad\qquad \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat}))) \\ \mathit{next}\ (e, [\,]) \qquad = \mathsf{inl}\ \star \\ \mathit{next}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (e + x, (e + x, \mathit{xs})), \end{array} \tag{71}$$

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 $\mathsf{in}_{\mathit{Nat}}\ (\mathsf{inl}\ \star)$. The definition of *diffs*,

$$\begin{array}{l} \mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat} \\ \mathit{diffs}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{step}\ (0, \mathit{xs}), \end{array} \tag{72}$$

with *step*,

$$\begin{array}{l} \mathit{step} : (\mathit{Nat} \times \mathit{ListNat}) \\ \qquad\qquad \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat}))) \\ \mathit{step}\ (e, [\,]) \qquad = \mathsf{inl}\ \star \\ \mathit{step}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (x - e, (x, \mathit{xs})), \end{array} \tag{73}$$

is arguably more natural than the previous one.

Now we can prove (54) 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],

$$\mathit{xs} = \mathit{ys} \quad \Leftrightarrow \quad \forall n \in \mathbb{N}.\ \ \mathit{approx}\ n\ \mathit{xs} = \mathit{approx}\ n\ \mathit{ys}, \tag{74}$$

where the function *approx* is defined by

$$\begin{array}{lll} \mathit{approx} \in \mathbb{N} \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] & & \\ \qquad\qquad \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] & & \\ \mathit{approx}\ 0 & \_ & = \bot \\ \mathit{approx}\ (n + 1)\ \bot & & = \bot \\ \mathit{approx}\ (n + 1)\ [\,] & & = [\,] \\ \mathit{approx}\ (n + 1)\ (x : \mathit{xs}) & & = x : \mathit{approx}\ n\ \mathit{xs}. \end{array} \tag{75}$$

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.

We have the following (yet again ignoring semantic brackets and contexts and also all uses of @):

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

$\forall n \in \mathbb{N}$. $\forall$ total xs containing total, finite numbers.

$\forall$ total and finite y.

approx n (*unfoldr step* $(y,$ *unfoldr next* $(y, xs))) =$

approx n xs.

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, whereas the second one requires a little more work:

approx $(k+1)$

(*unfoldr step* $(y,$ *unfoldr next* $(y, z : zs)))$

$=$ {definition of *unfoldr*, *next* and *step*}

approx $(k+1)$

$((y + z) - y :$

unfoldr step $(y + z,$ *unfoldr next* $(y + z, zs)))$

$=$ {$(y + z) - y = z$ for y, z total and finite}

approx $(k+1)$

$(z :$ *unfoldr step* $(y + z,$ *unfoldr next* $(y + z, zs)))$

$=$ {definition of *approx*}

$z :$ *approx* k

(*unfoldr step* $(y + z,$ *unfoldr next* $(y + z, zs)))$

$=$ {inductive hypothesis, $y + z$ is total and finite}

$z :$ *approx* k zs

$=$ {definition of *approx*}

approx $(k+1)$ $(z : zs)$.

Note that we need a lemma stating that $y + z$ is total and finite whenever y and z are.

10.3 Comparison

The last proof above, based on reasoning using domain-theoretic methods, is arguably more concise than the previous one, especially considering that it is more detailed. It also proves a stronger result, since it is not limited to finite lists.

In the short example in Section 2 we had to explicitly propagate preconditions through both *map* and *reverse*. In comparison, in this longer example we only had to propagate preconditions through $(+)$ (in the penultimate step of the last proof). Notice especially the second step of 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, for example, we want to process the result of *sums* using a fold, then 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.

11. Strict languages

We can treat strict languages (at least the somewhat odd language introduced below) using the framework developed so far by modelling strictness using seq, just like strict data type fields are handled in the Haskell Report [PJ03]. 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 $[\![\sigma]\!]_\perp = [\![\sigma]\!]$ for all types. For terms we let application be strict,

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

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 $[\![t]\!]_\perp\,\rho = [\![t]\!]\,\rho$.

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

$$t^* = \begin{cases} \mathsf{seq}\ {t_2}^*\ ({t_1}^*\ {t_2}^*), & t = t_1\ t_2, \\ \lambda x.{t_1}^*, & t = \lambda x.t_1, \\ t, & \text{otherwise.} \end{cases} \tag{78}$$

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 (45) and its corollary (46) that result from replacing $[\![\cdot]\!]$ with $[\![\cdot]\!]_\perp$.[1] The category-theoretic results from Section 8 immediately transfer to this new setting since the category is the same and $[\![t]\!]_\perp = [\![t]\!]$ for all closed terms t not containing applications.

12. 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 Smith [Smi84], who interprets a polymorphic variant of Martin-Löf type theory in an untyped language which shares many properties with our partial language $\mathcal{L}_2$; he does not consider coinductive types or seq, though. Beeson considers a variant of Martin-Löf type theory with W-types [Bee82]. W-types can be used to model strictly positive inductive and coinductive types [Dyb97, AAG05]. 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*) may 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.

[1] With modified preconditions: the *translations* of all terms have to belong to $\mathcal{L}_1'$. Unfortunately this was not mentioned in the published version of this paper.

Scott mentions that a category similar to $\mathsf{PER}_\sim$ is bicartesian closed [Sco76].

There is a vast body of literature 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. By writing down the results we make the details clear. Furthermore we apply the ideas to the problem of reasoning about programs, instead of using them only to interpret one theory in another. This is quite a natural idea, so it is not unreasonable to expect that others have made similar attempts. We know about [Dyb85], in which Dybjer explains how one can reason about an untyped partial language using total methods for expressions that are typeable (corresponding to our total values). We have not found any work that discusses fast and loose reasoning for strict languages.

Another angle on the work presented here is that we want to get around the fact that many category-theoretic isomorphisms are missing in categories like CPO. For instance, one cannot have a cartesian closed category with coproducts and fixpoints for all morphisms [HP90]. In this work we disallow all fixpoints except well-behaved ones that can be expressed as folds and unfolds. Another approach is to disallow recursion explicitly for sum types [BB91]. The MetaSoft project (see e.g. [BT83, Bli87]) advocated using "naive denotational semantics", a denotational semantics framework that does not incorporate reflexive domains. This means that some fixpoints (both on the type and the value level) are disallowed, with the aim that the less complex denotational structures may instead simplify understanding.

A case can be made for sometimes reasoning using a conservative approximate semantics, obtaining answers that are not always exactly correct [DJ04, discussion]. Programmers using languages like Haskell often ignore issues related to partiality anyway, so the spirit of many programs can be captured without treating all corner cases correctly. The methods described in this paper in a sense amount to using an approximate semantics, but with the ability to get exactly correct results by translating results involving $\sim$ to equalities with preconditions.

There is some correspondence between our approach and that of total and partial correctness reasoning for imperative programs, for example with Dijkstra's wp and wlp predicate transformers [Dij76]. In both cases, simpler approximate methods can be used to prove slightly weaker results than "what one really wants". However, in the w(l)p case, conjoining partial correctness with termination yields total correctness. In contrast, in our case, there is in general no (known) simple adjustment of a fast and loose proof to make a true proof of the same property. Nevertheless, the fast and loose proof already yields a true proof of a related property.

Sometimes it is argued that total functional programming should be used to avoid the problems with partial languages. Turner does that in the context of a language similar to the total one described here [Tur96], and discusses methods for circumventing the limitations associated with mandatory totality.

13. 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. Two total methods were described, one based on a set-theoretic semantics and one based on a bicartesian closed category, both using a partial equivalence relation to interpret results in the context of a domain-theoretic semantics.

We have focused on equational reasoning. However, note that, by adding and verifying some extra axioms, the category-theoretic approach can be used to reason about unequalities ($\neq$). Given this ability it should be possible to handle more complex logical formulas as well; we have not examined this possibility in detail, though. Since the method of Section 7 (without injective j) cannot handle unequalities, it is in some sense weaker.

The examples in Sections 2 and 10 indicate that total methods are sometimes 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 this, but can still make some remarks.

First it should be noted that $\sim$ is not a congruence: we can have $x \sim y$ but still have $f@x \not\sim f@y$ (if $f \notin \mathrm{dom}(\sim)$). 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 9. 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 leaves 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 $[\![t]\!] \in \mathrm{dom}(\sim)$. This would for example be the case if we implemented *foldr* (see Section 10.1) using fix instead of fold. Hence, if we explicitly prove that $[\![t]\!] \in \mathrm{dom}(\sim)$ then we can 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 leaves.

Now on to other issues. An obvious question is whether one can extend the results to more advanced languages incorporating stronger forms of recursive types, 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 treating a more advanced type system is that we want the results to be applicable to languages like Haskell, and matching more features of Haskell's type system is important for that goal. Still, the current results should be sufficient to reason about monomorphic Haskell programs using only polynomial recursive types, with one important caveat: 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, $[\![\mu(K_1 + Id)]\!]$ contains the different values *in* $(inl(\bot))$ and *in* $(inl(\star))$, but since the constructor *Zero* is nullary the Haskell data type *Nat* from Section 2 does not contain an analogue of *in* $(inl(\bot))$. One simple but unsatisfactory solution to this problem is to restrict the types used on the Haskell side to analogues of those discussed in this paper. Another approach is of course to rework the theory using sums-of-products

style data types. We foresee no major problems with this, but some details may require special attention.

Acknowledgments

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

References

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

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

[BB91] Marek A. Bednarczyk and Andrzej M. Borzyszkowski. Cpo's do not form a cpo and yet recursion works. In *VDM '91*, volume 551 of *LNCS*, pages 268–278. Springer-Verlag, 1991.

[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 *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.

[Bli87] Andrzej Blikle. *MetaSoft Primer, Towards a Metalanguage for Applied Denotational Semantics*, volume 288 of *LNCS*. Springer-Verlag, 1987.

[BT83] Andrzej Blikle and Andrzej Tarlecki. Naive denotational semantics. In *Information Processing 83*, pages 345–355. North-Holland, 1983.

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

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DJ04] Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In *MPC 2004*, volume 3125 of *LNCS*, pages 85–109. Springer-Verlag, 2004.

[Dyb85] Peter Dybjer. Program verification in a logical theory of constructions. In *FPCA'85*, volume 201 of *LNCS*, pages 334–349. Springer-Verlag, 1985. Appears in revised form as Programming Methodology Group Report 26, University of Göteborg and Chalmers University of Technology, 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, 1991.

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

[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, 2001.

[HP90] Hagen Huwig and Axel Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 73(1):101–112, 1990.

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

[JV04] Patricia Johann and Janis Voigtländer. Free theorems in the presence of *seq*. In *POPL'04*, pages 99–110. ACM Press, 2004.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA'91*, 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 *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 *Information Processing 83*, pages 513–523. Elsevier, 1983.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In *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, 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.

[Tur96] David Turner. Elementary strong functional programming. In *FPLE'95*, volume 1022 of *LNCS*, pages 1–13. Springer-Verlag, 1996.

[Wad89] Philip Wadler. Theorems for free! In *FPCA'89*, pages 347–359. ACM Press, 1989.