Submitted to ICFP'08

Free Theorems Involving Type Constructor Classes

Functional Pearl

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

Abstract

Free theorems are a charm, allowing the derivation of useful statements about programs from their (polymorphic) types alone. We show how to reap such theorems not only from polymorphism over ordinary types, but also from polymorphism over type *constructors* restricted by *class constraints*. Our prime application area are monads, which form the probably most popular type constructor class of Haskell. To demonstrate the broader scope, we also deal with a transparent way of introducing difference lists into a program, endowed with a neat and general correctness proof.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants

General Terms Languages, Verification

Keywords relational parametricity

1. Introduction

One of the strengths of functional languages like Haskell is an expressive type system. And yet, some of the benefits this should hold for reasoning about programs seem not to be realized to full extent. For example, Haskell uses monads (Moggi 1991) to structure programs by separating concerns (Wadler 1992; Liang et al. 1995) and to safely mingle pure and impure computations (Peyton Jones and Wadler 1993; Launchbury and Peyton Jones 1995). A lot of code can be kept independent of a concrete choice of monad. This pertains to standard Prelude functions like

$$sequence :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \rightarrow \mu\ [\alpha]\,,$$

but also to many user-defined functions. This is certainly a boon for modularity of programs. But also for reasoning?

Let us consider a more specific example, say functions of the type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \rightarrow \mu\ \mathsf{Int}$. Here are some:

$$f_1 = head$$

$$f_2\ ms = sequence\ ms \gg\!= return \circ sum$$

$$f_3 = f_2 \circ reverse$$

$$\begin{array}{lll} f_4\ [\,] & = return\ 0 \\ f_4\ (m : ms) & = \mathbf{do}\ \ i \leftarrow m \\ & \qquad\ \mathbf{let}\ l = length\ ms \\ & \qquad\ \mathbf{if}\ i > l\ \mathbf{then}\ return\ (i + l) \\ & \qquad\qquad\ \mathbf{else}\ f_4\ (drop\ i\ ms) \end{array}$$

As we see, there is quite a variety of such functions. There can be simple selection of one of the monadic computations from the input list (as in f_1), there can be sequencing of these monadic computations (in any order) and some action on the encapsulated values (as in f_2 and f_3), and the behaviour, in particular the choice which of the computations from the input list are actually performed, can even depend on the encapsulated values themselves (as in f_4, made a bit artificial here). Further possibilities are that some of the monadic computations from the input list are performed repeatedly, and so on. But still, all these functions also have something in common. They can only *combine* whatever monadic computations, and associated effects, they encounter in their input lists, but they cannot *introduce* new effects of any concrete monad, not even of the one they are actually operating on in a particular application instance. This is determined by the function type. For if an f were, on and of its own, to cause any additional effect to happen, be it by writing to the output, by introducing additional branching in the nondeterminism monad, or whatever, then it would immediately fail to get the above type parametric over μ. In a language like Haskell, should not we be able to profit from this kind of abstraction for reasoning purposes?

If so, what kind of insights can we hope for? One thing to expect is that in the special case when the concrete computations in an input list passed to an $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \rightarrow \mu\ \mathsf{Int}$ correspond to pure values (e.g., are values of type $\mathsf{IO}\ \mathsf{Int}$ that do not perform any actual input or output), then the same should hold of f's result for that input list. This is quite intuitive from the above observation about f being unable to cause new effects on its own. But what about more interesting statements, for example the preservation of certain invariants? Say, we pass to f a list of stateful computations and we happen to know that they do depend on, but do not alter (a certain part of) the state. Is this property preserved throughout the evaluation of a given f? Or say the effect encapsulated in f's input list is nondeterminism but we would like to simplify

the program by restricting the computation to a deterministically chosen representative from each nondeterministic manifold. Under what conditions, and for which kind of representative-selection functions, is this safe and does not lead to problems like a collapse of an erstwhile nonempty manifold to an empty one from which no representative can be chosen at all?

One could go and study these questions for particular functions like the f_1 to f_4 given further above. But instead we would like to answer them for any function of type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ in general, without consulting particular function definitions. And we would not like to restrict to the two or three scenarios depicted in the previous paragraph. Rather, we want to explore more abstract settings of which statements like the ones in question above can be seen, and dealt with, as particular instances. And, of course, we prefer a generic methodology that applies equally well to other types than the specific one of f considered so far in this introduction. These aims are not arbitrary or far-fetched. Precedent has been set with the theorems obtained for free by Wadler (1989) from relational parametricity (Reynolds 1983). Derivation of such free theorems, too, is a methodology applying not only to a single type, works independently of particular function definitions, and applies to a diverse range of scenarios: from simple algebraic laws to powerful program transformations (Gill et al. 1993), to meta-theorems about whole classes of algorithms (Voigtländer 2008b).

Unsurprisingly then, we do build on Reynolds' and Wadler's work. Of course, the framework that is usually considered when free theorems are derived needs to be extended to deal with types like $\mathsf{Monad}\ \mu \Rightarrow \ldots$. But the ideas needed to do so are there for the taking. Indeed, both relational parametricity extended for polymorphism over type constructors rather than over ordinary types only, as well as relational parametricity extended to take class constraints into account, are in the folklore. However, these two strands of possible extension have not been combined before, and not been used as we do here. Since we are most interested in demonstrating the prospects gained by that combination, we refrain from developing the folklore into a full-fledged formal apparatus that would stand to blur the intuitive ideas. This is not a theoretical paper. Also on purpose, we do not consider Haskell intricacies, like those studied by Johann and Voigtländer (2004) and Stenger and Voigtländer (2008), that do affect relational parametricity but in a way orthogonal to what is of interest here. Instead, we stay with Reynolds' and Wadler's simple model. For the sake of accessibility, we also stay close to Wadler's notation.

2. Free Theorems, in Full Beauty

The key to deriving free theorems is to interpret types as relations. For example, given a type signature $f :: [\alpha] \to [\alpha]$, we take the type and replace every quantification over type variables, including implicit quantification (note that the type $[\alpha] \to [\alpha]$, by Haskell convention, really means $\forall\alpha.\ [\alpha] \to [\alpha]$), by quantification over relation variables: $\forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$. Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like Int are read as identity relations,
- for relations $\mathcal{R}$ and $\mathcal{S}$, we have

  $$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}.\ (f\ a, g\ b) \in \mathcal{S}\}\,,$$

  and
- for types τ and τ' with at most one free variable, say α, and a function $\mathcal{F}$ on relations such that every relation $\mathcal{R}$ between closed types τ_1 and τ_2, denoted $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, is mapped to a relation $\mathcal{F}\,\mathcal{R} : \tau[\tau_1/\alpha] \Leftrightarrow \tau'[\tau_2/\alpha]$, we have

  $$\forall\mathcal{R}.\,\mathcal{F}\,\mathcal{R} = \{(u, v) \mid \forall\tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\,\mathcal{R}\}\,.$$

  (Here, u_{τ_1} is the instantiation of $u :: \forall\alpha.\ \tau$ to the type τ_1, and similarly for v_{τ_2}. In what follows, we will always leave type instantiation implicit.)

Also, every fixed type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to the relation $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$ defined by

$$[\mathcal{R}] = \{([\,], [\,])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R},\ (as, bs) \in [\mathcal{R}]\}\,,$$

the Maybe type constructor maps every relation $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to the relation $\mathsf{Maybe}\,\mathcal{R} : \mathsf{Maybe}\ \tau_1 \Leftrightarrow \mathsf{Maybe}\ \tau_2$ defined by

$$\begin{aligned}\mathsf{Maybe}\,\mathcal{R} = {} & \{(\mathsf{Nothing}, \mathsf{Nothing})\} \cup \\ & \{(\mathsf{Just}\ a, \mathsf{Just}\ b) \mid (a, b) \in \mathcal{R}\}\,,\end{aligned}$$

and similarly for other user-definable types.

The key insight of relational parametricity à la Reynolds (1983) now is that any expression over relations that can be built as above, by interpreting a closed type, denotes the identity relation on that type.

For the above example, this means that any $f :: \forall\alpha.\ [\alpha] \to [\alpha]$ satisfies $(f, f) \in \forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$, which by unfolding some of the above definitions is equivalent to having for every τ_1, τ_2, $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, $l :: [\tau_1]$, and $l' :: [\tau_2]$ that $(l, l') \in [\mathcal{R}]$ implies $(f\ l, f\ l') \in [\mathcal{R}]$, or, specialised to the function level, for every $h :: \tau_1 \to \tau_2$ and $l :: [\tau_1]$ that $f\ (map\ h\ l) = map\ h\ (f\ l)$.

When we want to extend the treatment to type constructor classes, we have to deal with two new aspects: with quantification over type *constructor* variables (rather than just over type variables) and with *class constraints* (Wadler and Blott 1989). For both aspects, the required extensions to the interpretation of types as relations appear to be folklore, but have seldomly been spelled out and have not been put to use before as we do in this paper.

Regarding quantification over type *constructor* variables, the necessary adaptation is as follows. Just as free type variables are interpreted as relations between arbitrarily chosen closed types (and then quantified over via relation variables), free type constructor variables are interpreted as functions on such relations tied to arbitrarily chosen type constructors. Formally, let κ_1 and κ_2 be type constructors (of kind $* \to *$). A *relational action* for them, denoted $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$, is a function $\mathcal{F}$ on relations between closed types such that every $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ (for arbitrary τ_1 and τ_2) is mapped to an $\mathcal{F}\,\mathcal{R} : \kappa_1\ \tau_1 \Leftrightarrow \kappa_2\ \tau_2$. For example, the function $\mathcal{F}$ that maps every $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to

$$\begin{aligned}\mathcal{F}\,\mathcal{R} = {} & \{(\mathsf{Nothing}, [\,])\} \cup \\ & \{(\mathsf{Just}\ a, b : bs) \mid (a, b) \in \mathcal{R},\ bs :: [\tau_2]\}\end{aligned}$$

is a relational action $\mathcal{F} : \mathsf{Maybe} \Leftrightarrow []$. The relational interpretation of a type quantifying over a type constructor variable is now performed in an analogous way as explained for quantification over type (and then, relation) variables at the beginning of this section. In different formulations and detail, the same basic idea is mentioned or used by Fegaras and Sheard (1996), Kučan (1997), Takeuti (2001), and Vytiniotis and Weirich (2007).

Regarding *class constraints*, Wadler (1989, Section 3.4) directs the way by explaining how to treat the type class Eq in the context of deriving free theorems. The idea is to simply restrict the relations chosen as interpretation for type variables that are subject to a class constraint. Clearly, only relations between types that are instances of the class under consideration are allowed. Further restrictions are obtained from the respective class declaration. Namely, the restrictions must precisely ensure that every class method is related to itself by the relational interpretation of its type. This then guarantees that the overall result (i.e., that the relational interpretation of every closed type is an identity relation) stays intact. The same approach immediately applies to type constructor classes as well.

Consider, for example, the Monad class declaration:

$$\begin{array}{l}\textbf{class}\ \mathsf{Monad}\ \mu\ \textbf{where}\\ \quad return :: \alpha \to \mu\ \alpha\\ \quad (\gg\!=) :: \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta\end{array}$$

Since the type of $return$ is $\forall\mu.\ \mathsf{Monad}\ \mu \Rightarrow (\forall\alpha.\ \alpha \to \mu\ \alpha)$, we expect that $(return, return) \in \forall\mathcal{F}.\ \mathsf{Monad}\ \mathcal{F} \Rightarrow (\forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, and similarly for $\gg\!=$. The constraint "Monad $\mathcal{F}$" on a relational action is now defined in precisely such a way that both conditions will be fulfilled.

Definition 1. Let κ_1 and κ_2 be type constructors that are instances of Monad and let $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ be a relational action. If

- $(return_{\kappa_1}, return_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R}$ and
- $((\gg\!=_{\kappa_1}), (\gg\!=_{\kappa_2})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$,

then $\mathcal{F}$ is called a Monad-*action*. (While we have decided to generally leave type instantiation implicit, we explicitly retain instantiation of type constructors in what follows, except for some examples.)

For example, given the following standard Monad instance definitions:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Monad\ Maybe}\ \textbf{where}\\ \quad return\ a = \mathsf{Just}\ a\\ \quad \mathsf{Nothing} \gg\!= k = \mathsf{Nothing}\\ \quad \mathsf{Just}\ a \quad \gg\!= k = k\ a\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathsf{Monad}\ []\ \textbf{where}\\ \quad return\ a = [a]\\ \quad as \gg\!= k = concat\ (map\ k\ as)\end{array}$$

the relational action $\mathcal{F} : \mathsf{Maybe} \Leftrightarrow []$ given above is *not* a Monad-action, because it is not the case that $((\gg\!=_{\mathsf{Maybe}}), (\gg\!=_{[]})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$. To see this, consider $\mathcal{R} = \mathcal{S} = id_{\mathsf{Int}}$, $m_1 = \mathsf{Just}\ 1$, $m_2 = [1, 2]$, $k_1 = \lambda i \to \textbf{if}\ i > 1\ \textbf{then}\ \mathsf{Just}\ i\ \textbf{else}\ \mathsf{Nothing}$, and $k_2 = \lambda i \to reverse\ [2..i]$. Clearly, $(m_1, m_2) \in \mathcal{F}\ id_{\mathsf{Int}}$ and $(k_1, k_2) \in id_{\mathsf{Int}} \to \mathcal{F}\ id_{\mathsf{Int}}$, but $(m_1 \gg\!=_{\mathsf{Maybe}} k_1, m_2 \gg\!=_{[]} k_2) = (\mathsf{Nothing}, [2]) \notin \mathcal{F}\ id_{\mathsf{Int}}$.

We are now ready to derive free theorems involving (polymorphism over) type constructor classes. For example, functions $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ as considered in the introduction will necessarily always satisfy $(f, f) \in \forall\mathcal{F}.\ \mathsf{Monad}\ \mathcal{F} \Rightarrow [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$, i.e., for every choice of type constructors κ_1 and κ_2 that are instances of Monad, and every Monad-action $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$, we have $(f_{\kappa_1}, f_{\kappa_2}) \in [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$. In the next section we prove several theorems by instantiating the $\mathcal{F}$ here, and provide plenty examples of interesting results obtained for concrete monads.

3. One Application: Reasoning about Monadic Programs

For most of this section, we focus on functions $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$. However, it should be emphasised that results of the same spirit can be systematically obtained for other types involving quantification over Monad-restricted type constructor variables just as well.

As mentioned in the introduction, one first intuitive statement we naturally expect to hold is that when all the monadic values supplied to f in the input list are actually pure (not associated with any proper monadic effect), then f's result value, though of some monadic type, should also be pure. After all, f itself, being polymorphic over μ, cannot introduce effects from any specific monad. This statement is expected to hold no matter what monad the input values live in. For example, if the input list consists of computations in the list monad, defined in the previous section and modelling nondeterminism, but all the concretely passed values actually correspond to deterministic computations, then we expect that f's result value also corresponds to a deterministic computation. Similarly, if the input list consists of IO computations, but we only pass ones that happen to have no side-effect at all, then f's result, though living in the IO monad, should also be side-effect-free. To capture the notion of "purity" independently of any concrete monad, we use the convention that the pure computations in any monad are those that may be the result of a call to $return$. Note that this does not mean that the values in the input list must *syntactically* be $return$-calls. Rather, each of them only needs to be *semantically equivalent* to some such call. The desired statement is now formalised, and proved, as follows.

Theorem 1. *Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, let κ be an instance of* Monad*, and let $l :: [\kappa\ \mathsf{Int}]$. If every element in l is a $return_\kappa$-image, then so is $f_\kappa\ l$.*

Proof. We prove that for every $l' :: [\mathsf{Int}]$,

$$f_\kappa\ (map\ return_\kappa\ l') = return_\kappa\ (\mathsf{unId}\ (f_{\mathsf{Id}}\ (map\ \mathsf{Id}\ l'))),$$

where

$$\textbf{newtype}\ \mathsf{Id}\ \alpha = \mathsf{Id}\ \{\mathsf{unId} :: \alpha\}$$

$$\begin{array}{l}\textbf{instance}\ \mathsf{Monad\ Id}\ \textbf{where}\\ \quad return\ a = \mathsf{Id}\ a\\ \quad \mathsf{Id}\ a \gg\!= k = k\ a\end{array}$$

To do so, we first show that $\mathcal{F} : \kappa \Leftrightarrow \mathsf{Id}$ with

$$\mathcal{F}\ \mathcal{R} = return_\kappa^{-1}\ ;\ \mathcal{R}\ ;\ \mathsf{Id},$$

where ";" is relation composition and "$^{-1}$" gives the inverse of a function graph, is a Monad-action. Indeed,

- $(return_\kappa, return_{\mathsf{Id}}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R}$, since for every $\mathcal{R}$ and $(a, b) \in \mathcal{R}$, $(return_\kappa\ a, return_{\mathsf{Id}}\ b) = (return_\kappa\ a, \mathsf{Id}\ b) \in return_\kappa^{-1}\ ;\ \mathcal{R}\ ;\ \mathsf{Id}$, and
- $((\gg\!=_\kappa), (\gg\!=_{\mathsf{Id}})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$, since for every $\mathcal{R}$, $\mathcal{S}$, $(a, b) \in \mathcal{R}$, and $(k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$, $(return_\kappa\ a \gg\!=_\kappa k_1, \mathsf{Id}\ b \gg\!=_{\mathsf{Id}} k_2) = (k_1\ a, k_2\ b) \in \mathcal{F}\ \mathcal{S}$. (Note the use of a monad law for κ.)

Hence, by what we derived at the end of the previous section, $(f_\kappa, f_{\mathsf{Id}}) \in [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$. Given that we have $\mathcal{F}\ id_{\mathsf{Int}} = return_\kappa^{-1}\ ;\ \mathsf{Id} = (return_\kappa \circ \mathsf{unId})^{-1}$, this implies the claim.

We can now reason for specific monads as follows.

Example 1. Let $l :: [[\mathsf{Int}]]$, i.e., $l :: [\kappa\ \mathsf{Int}]$ for $\kappa = []$. We might be interested in establishing that when every element in l is (evaluated to) a singleton list, then the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l will be a singleton list as well. While this is easy to see for f_1, f_2, and f_3 from the introduction, it is maybe not so immediately obvious for the f_4 given there. However, Theorem 1 tells us without any further effort that the statement in question does indeed hold for f_4, and for any other f of the same type.

Likewise, we obtain the statement about side-effect-free computations in the IO monad envisaged above. All we rely on then is that the IO monad, like the list monad, satisfies the monad law that $return\ a \gg\!= k$ is $k\ a$, as used in the proof of Theorem 1.

A second general statement we are interested in is to deal with the case that the monadic computations provided as input are not necessarily pure, but we have a way of discarding the monadic layer and recovering underlying values. Somewhat akin to $unsafePerformIO :: \mathsf{IO}\ \alpha \to \alpha$, but for other monads and hopefully safe. Then, if we are interested only in a thus projected result value of f, can we show that it only depends on likewise projected input values, i.e., that we can discard any effects from the monadic computations in f's input list when we are not interested in the effectful part of the output computation? Clearly, it would be too much to expect this to work for arbitrary "projections", or even arbitrary monads. Rather, we need to devise appropriate restrictions and prove that they suffice. The formal statement turns out to be as follows.

Theorem 2. *Let* $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, *let* κ *be an instance of* Monad, *and let* $p :: \kappa\ \alpha \to \alpha$. *If*

- $p \circ return_\kappa = id$ *and*
- *for every choice of closed types* τ *and* τ', $m :: \kappa\ \tau$, *and* $k :: \tau \to \kappa\ \tau'$,

$$p\ (m \gg\!=_\kappa k) = p\ (k\ (p\ m))\,,$$

then $p \circ f_\kappa$ *gives the same result for any two lists of same length whose corresponding elements have the same* p*-images, i.e.,* $p \circ f_\kappa$ *can be "factored" as* $g \circ (map\ p)$ *for some suitable* $g :: [\mathsf{Int}] \to \mathsf{Int}$.

Proof. We prove that for every $l :: [\kappa\ \mathsf{Int}]$,

$$p\ (f_\kappa\ l) = \mathsf{unId}\ (f_{\mathsf{Id}}\ (map\ (\mathsf{Id} \circ p)\ l))\,,$$

where the type constructor Id and its Monad instance definition are as in the proof of Theorem 1. To do so, we first show that $\mathcal{F} : \kappa \Leftrightarrow \mathsf{Id}$ with

$$\mathcal{F}\ \mathcal{R} = p\ ;\ \mathcal{R}\ ;\ \mathsf{Id}$$

is a Monad-action. Indeed,

- $(return_\kappa, return_{\mathsf{Id}}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R}$, since for every $\mathcal{R}$ and $(a, b) \in \mathcal{R}$, $(return_\kappa\ a, b) \in p\ ;\ \mathcal{R}$ by $p\ (return_\kappa\ a) = a$, and
- $((\gg\!=_\kappa), (\gg\!=_{\mathsf{Id}})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$, since for every $\mathcal{R}$, $\mathcal{S}$, $(m, b) \in p\ ;\ \mathcal{R}$, and $(k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$, $(m \gg\!=_\kappa k_1, \mathsf{Id}\ b \gg\!=_{\mathsf{Id}} k_2) \in p\ ;\ \mathcal{S}$; Id by $p\ (m \gg\!=_\kappa k_1) = p\ (k_1\ (p\ m))$ and $(k_1\ (p\ m), k_2\ b) \in p\ ;\ \mathcal{S}$; Id (which holds due to $(k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$ and $(p\ m, b) \in \mathcal{R}$).

Hence, $(f_\kappa, f_{\mathsf{Id}}) \in [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$. Given that we have $\mathcal{F}\ id_{\mathsf{Int}} = p\ ;\ \mathsf{Id} = \mathsf{Id} \circ p = p\ ;\ \mathsf{unId}^{-1}$, this implies the claim.

Note that no monad laws at all are needed in the above proof. The same will be true for the other theorems we prove in this section, except for Theorem 5. But first, we consider several example applications of Theorem 2.

Example 2. Consider the well-known writer, or logging, monad (specialised here to the String monoid):

```
newtype Writer α = Writer (α, String)

instance Monad Writer where
  return a = Writer (a, "")
  Writer (a, s) >>= k =
    Writer (case k a of Writer (a', s') → (a', s ++ s'))
```

Assume we are interested in applying an $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to an $l :: [\mathsf{Writer\ Int}]$, yielding a monadic result of type Writer Int. Assume further that for some particular purpose during reasoning about the overall program, we are only interested in the actual integer value encapsulated in that result, as extracted by the following function:

$$p :: \mathsf{Writer}\ \alpha \to \alpha$$
$$p\ (\mathsf{Writer}\ (a, s)) = a$$

Intuition suggests that then the value of $p\ (f\ l)$ should not depend on any logging activity of elements in l. That is, if l were replaced by another $l' :: [\mathsf{Writer\ Int}]$ encapsulating the same integer values, put potentially attached with different logging information, then $p\ (f\ l')$ should give exactly the same value. Since the given p fulfils the required conditions, Theorem 2 confirms this intuition.

It should also be instructive here to consider a negative example.

Example 3. Recall the list monad defined in the previous section. It is tempting to use $head :: [\alpha] \to \alpha$ as an extraction function and expect that for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, we can factor $head \circ f$ as $g \circ (map\ head)$ for some suitable $g :: [\mathsf{Int}] \to \mathsf{Int}$. But actually this fails in a subtle way. Consider, for example, the (for the sake of simplicity, artificial) function

$$\begin{aligned} &f_5 :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int} \\ &f_5\ [\,] \quad\quad\quad = return\ 0 \\ &f_5\ (m : ms) = \mathbf{do}\ \ i \leftarrow m \\ &\qquad\qquad\qquad\quad\ \ f_5\ (\mathbf{if}\ i > 0\ \mathbf{then}\ ms\ \mathbf{else}\ tail\ ms) \end{aligned}$$

Then for $l = [[1], [\,]]$ and $l' = [[1, 0], [\,]]$, both of type $[[\mathsf{Int}]]$, we have $map\ head\ l = map\ head\ l'$, but $head\ (f_5\ l) \neq head\ (f_5\ l')$. In fact, the left-hand side of this inequation leads to an "head of empty list"-error, whereas the right-hand side delivers the value 0. Clearly, this means that the supposed g cannot exist for f_5 and $head$.

An explanation for the observed failure is provided by the conditions imposed on p in Theorem 2. It is simply not true that for every m and k, $head\ (m \gg\!= k) = head\ (k\ (head\ m))$. More concretely, the failure for f_5 observed above arises from this equation being violated for $m = [1, 0]$ and $k = \lambda i \to \mathbf{if}\ i > 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ [0]$.

Since the previous (counter-)example is a bit peculiar in its reliance on runtime errors, let us consider a related setting without empty lists, an example also serving to further emphasise the predictive power of the conditions on p in Theorem 2.

Example 4. Assume, just for the scope of this example, that the type constructor [] yields (the types of) nonempty lists only. Clearly, it becomes an instance of Monad by just the same definition as given in Section 2. There are now several choices for a never failing extraction function $p :: [\alpha] \to \alpha$. For example, p could be $head$, could be $last$, or could be the function that always returns the element in the middle position of its input list (and, say, the left one of the two middle elements in the case of a list of even length). But which of these candidates are "good" in the sense of providing, for

every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, a factorisation of $p \circ f$ into $g \circ (map\ p)$? The answer is provided by the two conditions on p in Theorem 2, which specialised to the (nonempty) list monad require that

- for every a, $p\ [a] = a$, and
- for every choice of closed types τ and τ', $m :: [\tau]$, and $k :: \tau \to [\tau']$, $p\ (concat\ (map\ k\ m)) = p\ (k\ (p\ m))$.

From these conditions it is easy to see that now $p = head$ is good (in contrast to the situation in Example 3), and so is $p = last$, while the proposed "middle extractor" is not. It does not fulfil the second condition above, roughly because k does not necessarily map all its inputs to equally long lists. (A concrete counterexample f_6, of appropriate type, can easily be produced from this observation.)

Next, we would like to tackle reasoning not about the complete absence of (à la Theorem 1), or disregard for (à la Theorem 2), monadic effects, but about finer nuances. Often, we know certain computations to realize only some of the potential effects to which they would be entitled according to the monad they live in. If, for example, the effect under consideration is nondeterminism à la the standard list monad, then we might know of some computations in that monad that they realize only none-or-one-nondeterminism, i.e., never produce more than one answer, but may produce none at all. Or we might know that they realize only non-failing-nondeterminism, i.e., always produce at least one answer, but may produce more than one. Then, we might want to argue that the respective nature of nondeterminism is preserved when combining such computations using, say, a function $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$. This would mean that applying any such f to any list of empty-or-singleton lists always gives an empty-or-singleton list as result, and that applying any such f to any list of nonempty lists only gives a nonempty list as result for sure. Or, in the case of an exception monad (Either String), we might want to establish that an application of f cannot possibly lead to any exceptional value (error description string) other than those already present somewhere in its input list. Such "invariants" can often be captured by identifying a certain "subspace" of the monadic type in question that forms itself a monad, or, indeed, by "embedding" another, "smaller", monad into the one of interest. Formal counterparts of the intuition behind the previous sentence and the vague phrases occurring therein can be found in the following definition and theorem, as well as in the subsequent examples.

Definition 2. Let κ_1 and κ_2 be instances of Monad and let $h :: \kappa_2\ \alpha \to \kappa_1\ \alpha$. If

- $h \circ return_{\kappa_2} = return_{\kappa_1}$ and
- for every choice of closed types τ and τ', $m :: \kappa_2\ \tau$, and $k :: \tau \to \kappa_2\ \tau'$,

$$h\ (m \gg\!=_{\kappa_2} k) = h\ m \gg\!=_{\kappa_1} h \circ k\,,$$

then h is called a *monad morphism*.

Theorem 3. *Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, let $h :: \kappa_2\ \alpha \to \kappa_1\ \alpha$ be a monad morphism, and let $l :: [\kappa_1\ \mathsf{Int}]$. If every element in l is an h-image, then so is $f_{\kappa_1}\ l$.*

Proof. We prove that for every $l' :: [\kappa_2\ \mathsf{Int}]$,

$$f_{\kappa_1}\ (map\ h\ l') = h\ (f_{\kappa_2}\ l')\,. \tag{1}$$

To do so, we first show that $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ with

$$\mathcal{F}\ \mathcal{R} = (\kappa_1\ \mathcal{R})\ ;\ h^{-1}$$

is a Monad-action. Indeed,

- $(return_{\kappa_1}, return_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R}$, since for every $\mathcal{R}$ and $(a, b) \in \mathcal{R}$, $(return_{\kappa_1}\ a, h\ (return_{\kappa_2}\ b)) = (return_{\kappa_1}\ a, return_{\kappa_1}\ b) \in \kappa_1\ \mathcal{R}$ by $(return_{\kappa_1}, return_{\kappa_1}) \in \forall\mathcal{R}.\ \mathcal{R} \to \kappa_1\ \mathcal{R}$ (which holds due to $return_{\kappa_1} :: \forall\alpha.\ \alpha \to \kappa_1\ \alpha$), and
- $((\gg\!=_{\kappa_1}), (\gg\!=_{\kappa_2})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{F}\ \mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S})$, since for every $\mathcal{R}$, $\mathcal{S}$, $(m_1, m_2) \in (\kappa_1\ \mathcal{R})\ ;\ h^{-1}$, and $(k_1, k_2) \in \mathcal{R} \to ((\kappa_1\ \mathcal{S})\ ;\ h^{-1})$,

  $$\begin{array}{l}(m_1 \gg\!=_{\kappa_1} k_1, h\ (m_2 \gg\!=_{\kappa_2} k_2)) = \\ (m_1 \gg\!=_{\kappa_1} k_1, h\ m_2 \gg\!=_{\kappa_1} h \circ k_2) \in \kappa_1\ \mathcal{S}\end{array}$$

  by $((\gg\!=_{\kappa_1}), (\gg\!=_{\kappa_1})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \kappa_1\ \mathcal{R} \to ((\mathcal{R} \to \kappa_1\ \mathcal{S}) \to \kappa_1\ \mathcal{S})$, $(m_1, h\ m_2) \in \kappa_1\ \mathcal{R}$, and $(k_1, h \circ k_2) \in \mathcal{R} \to \kappa_1\ \mathcal{S}$.

Hence, $(f_{\kappa_1}, f_{\kappa_2}) \in [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$. Given that we have $\mathcal{F}\ id_{\mathsf{Int}} = (\kappa_1\ id_{\mathsf{Int}})\ ;\ h^{-1} = h^{-1}$, this implies the claim. (Note that $\kappa_1\ id_{\mathsf{Int}}$ is the relational interpretation of the closed type $\kappa_1\ \mathsf{Int}$, and thus itself denotes $id_{\kappa_1\ \mathsf{Int}}$.)

Using Theorem 3, we can indeed prove the statements mentioned for the list and exception monads above Definition 2. Here, for diversion, we instead prove some results about more stateful computations.

Example 5. Consider the well-known reader monad:

$$\begin{array}{l}\mathbf{newtype}\ \mathsf{Reader}\ \rho\ \alpha = \mathsf{Reader}\ (\rho \to \alpha) \\ \\ \mathbf{instance}\ \mathsf{Monad}\ (\mathsf{Reader}\ \rho)\ \mathbf{where} \\ \quad return\ a = \mathsf{Reader}\ (\lambda r \to a) \\ \quad \mathsf{Reader}\ g \gg\!= k = \\ \qquad \mathsf{Reader}\ (\lambda r \to \mathbf{case}\ k\ (g\ r)\ \mathbf{of}\ \mathsf{Reader}\ g' \to g'\ r)\end{array}$$

Assume we are given a list of computations in a Reader monad, but it happens that all present computations depend only on a certain part of the environment type. For example, for some closed types τ_1 and τ_2, $l :: [\mathsf{Reader}\ (\tau_1, \tau_2)\ \mathsf{Int}]$, and for every element $\mathsf{Reader}\ g$ in l, $g\ (x, y)$ never depends on y. We come to expect that the same kind of independence should then hold for the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l. And indeed this holds by Theorem 3 with the following monad morphism:

$$\begin{array}{l}h :: \mathsf{Reader}\ \tau_1\ \alpha \to \mathsf{Reader}\ (\tau_1, \tau_2)\ \alpha \\ h\ (\mathsf{Reader}\ g) = \mathsf{Reader}\ (g \circ fst)\end{array}$$

It is also possible to connect more different monads, even involving the IO monad.

Example 6. Let $l :: [\mathsf{IO}\ \mathsf{Int}]$ and assume that the only side-effects that elements in l have consist of writing strings to the output. We would like to use Theorem 3 to argue that the same is then true for the result of applying any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to l. To this end, we need to somehow capture the concept of "writing (potentially empty) strings to the output as only side-effect of an IO computation" via an embedding from another monad. Quite naturally, we reuse the Writer monad from Example 2. The embedding function is as

follows:

$$h :: \mathsf{Writer}\ \alpha \to \mathsf{IO}\ \alpha$$
$$h\ (\mathsf{Writer}\ (a, s)) = putStr\ s \gg return\ a$$

What is left to do is to show that h is a monad morphism. But this follows from $putStr$ "" $= return\ ()$, $putStr\ (s +\!\!+ s') = putStr\ s \gg putStr\ s'$, and monad laws for the IO monad.

Similarly to the above, it would also be possible to show that when the IO computations in l do only read from the input (via, possibly repeated, calls to $getChar$), then the same is true of $f\ l$. Instead of exercising this through, we turn to general state transformers.

Example 7. Consider the well-known state monad:

$$\mathbf{newtype}\ \mathsf{State}\ \sigma\ \alpha = \mathsf{State}\ (\sigma \to (\alpha, \sigma))$$

$$\begin{aligned}&\mathbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ \sigma)\ \mathbf{where}\\ &\quad return\ a = \mathsf{State}\ (\lambda s \to (a, s))\\ &\quad \mathsf{State}\ g \gg\!= k =\\ &\qquad \mathsf{State}\ (\lambda s \to \mathbf{let}\ (a, s') = g\ s\ \mathbf{in}\\ &\qquad\qquad \mathbf{case}\ k\ a\ \mathbf{of}\ \mathsf{State}\ g' \to g'\ s')\end{aligned}$$

Intuitively, this extends the reader monad by not only allowing a computation to depend on an input state, but also to transform the state to be passed to a subsequent computation. A natural question now is whether being a specific state transformer that actually corresponds to a read-only computation is an invariant that is preserved when computations are combined. That is, given some closed type τ and $l :: [\mathsf{State}\ \tau\ \mathsf{Int}]$ such that for every element $\mathsf{State}\ g$ in l, $snd \circ g = id$, is it the case that for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, also $f\ l$ is of the form $\mathsf{State}\ g$ for some g with $snd \circ g = id$? The positive answer is provided by Theorem 3 with the following monad morphism:

$$h :: \mathsf{Reader}\ \tau\ \alpha \to \mathsf{State}\ \tau\ \alpha$$
$$h\ (\mathsf{Reader}\ g) = \mathsf{State}\ (\lambda s \to (g\ s, s))$$

Similarly to the above, we can show preservation of the invariant that a computation transforms the state "in the background", while the primary result value is independent of the input state. That is, if for every element $\mathsf{State}\ g$ in l, there exists an $i :: \mathsf{Int}$ with $fst \circ g = const\ i$, then the same applies to $f\ l$. It should also be possible to transfer the above kind of reasoning to the ST monad (Launchbury and Peyton Jones 1995).

As a final statement about our pet type, $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, we would like to show that we can abstract from some aspects of the effectful computations in the input list if we are interested in the effects of the final result only up to the same abstraction. For conveying between the full effect space and its abstraction, we again use monad morphisms.

Theorem 4. *Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ and let $h :: \kappa_2\ \alpha \to \kappa_1\ \alpha$ be a monad morphism. Then $h \circ f_{\kappa_2}$ gives the same result for any two lists of same length whose corresponding elements have the same h-images.*

Proof. Let $l_1, l_2 :: [\kappa_2\ \mathsf{Int}]$ be such that $map\ h\ l_1 = map\ h\ l_2$. Then $h\ (f_{\kappa_2}\ l_1) = h\ (f_{\kappa_2}\ l_2)$ by statement (1) from the proof of Theorem 3.

Example 8. Consider the well-known exception monad:

$$\begin{aligned}&\mathbf{instance}\ \mathsf{Monad}\ (\mathsf{Either\ String})\ \mathbf{where}\\ &\quad return\ a = \mathsf{Right}\ a\\ &\quad \mathsf{Left}\ err \gg\!= k = \mathsf{Left}\ err\\ &\quad \mathsf{Right}\ a \gg\!= k = k\ a\end{aligned}$$

We would like to argue that if we are only interested in whether the result of f for some input list over the type $\mathsf{Either\ String\ Int}$ is an exceptional value or not (and which ordinary value is encapsulated in the latter case), but do not care what the concrete error description string is in the former case, then the answer is independent of the concrete error description strings potentially appearing in the input list. Formally, let $l_1, l_2 :: [\mathsf{Either\ String\ Int}]$ be of same length, and let corresponding elements either be both tagged with Left (but not necessarily containing the same strings) or be identical Right-tagged values. Then for every $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, $f\ l_1$ and $f\ l_2$ either are both tagged with Left or are identical Right-tagged values. This holds by Theorem 4 with the following monad morphism:

$$h :: \mathsf{Either\ String}\ \alpha \to \mathsf{Maybe}\ \alpha$$
$$h\ (\mathsf{Left}\ err) = \mathsf{Nothing}$$
$$h\ (\mathsf{Right}\ a) = \mathsf{Just}\ a$$

Just to reinforce that our approach is not specific to our pet type alone, we end this section by giving a theorem obtained for another type, the one of $sequence$, also showing that mixed quantification over type constructor variables and ordinary type variables can very well be handled.

Theorem 5. *Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \to \mu\ [\alpha]$ and let $h :: \kappa_2\ \alpha \to \kappa_1\ \alpha$ be a monad morphism. Then for every choice of closed types τ_1 and τ_2, $g :: \tau_1 \to \tau_2$, $l_1 :: [\kappa_1\ \tau_1]$, and $l_2 :: [\kappa_2\ \tau_1]$, $map\ (\gg\!=_{\kappa_1}\ return_{\kappa_1})\ l_1 = map\ h\ l_2$ implies*

$$f_{\kappa_1}\ (map\ (\gg\!=_{\kappa_1}\ return_{\kappa_1} \circ g)\ l_1) \gg\!=_{\kappa_1}\ return_{\kappa_1} = h\ (f_{\kappa_2}\ l_2) \gg\!=_{\kappa_1}\ return_{\kappa_1} \circ map\ g\,.$$

Using the function

$$fmap :: \mathsf{Monad}\ \mu \Rightarrow (\alpha \to \beta) \to \mu\ \alpha \to \mu\ \beta$$
$$fmap\ g\ m = m \gg\!= return \circ g$$

and a monad law for κ_1, this can be simplified as follows:

$$f_{\kappa_1} \circ map\ (fmap_{\kappa_1}\ g) \circ map\ h = fmap_{\kappa_1}\ (map\ g) \circ h \circ f_{\kappa_2}\,.$$

4. Another Application: Difference Lists, Transparently

It is a well-known problem that computations over lists sometimes suffer from a quadratic runtime blow-up due to left-associatively nested appends. For example, this is the case for flattening a tree of

the type

$$\mathbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Leaf}\ \alpha \mid \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)$$

using the following function:

$$\begin{aligned} &\mathit{flatten} :: \mathsf{Tree}\ \alpha \to [\alpha] \\ &\mathit{flatten}\ (\mathsf{Leaf}\ a) \quad\ = [a] \\ &\mathit{flatten}\ (\mathsf{Node}\ t_1\ t_2) = \mathit{flatten}\ t_1 +\!\!+ \mathit{flatten}\ t_2 \end{aligned}$$

An equally well-known solution is to switch to an alternative representation of lists as functions, by abstraction over the list end, often called difference lists. In the formulation of Hughes (1986), but encapsulated as an explicitly new data type:

$$\begin{aligned} &\mathbf{newtype}\ \mathsf{DList}\ \alpha = \mathsf{DL}\ \{\mathsf{unDL} :: [\alpha] \to [\alpha]\} \\ \\ &\mathit{rep} :: [\alpha] \to \mathsf{DList}\ \alpha \\ &\mathit{rep}\ l = \mathsf{DL}\ (l +\!\!+) \\ \\ &\mathit{abs} :: \mathsf{DList}\ \alpha \to [\alpha] \\ &\mathit{abs}\ (\mathsf{DL}\ f) = f\ [\,] \\ \\ &\mathit{emptyR} :: \mathsf{DList}\ \alpha \\ &\mathit{emptyR} = \mathsf{DL}\ \mathit{id} \\ \\ &\mathit{consR} :: \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\ &\mathit{consR}\ a\ (\mathsf{DL}\ f) = \mathsf{DL}\ ((a :) \circ f) \\ \\ &\mathit{appendR} :: \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\ &\mathit{appendR}\ (\mathsf{DL}\ f)\ (\mathsf{DL}\ g) = \mathsf{DL}\ (f \circ g) \end{aligned}$$

Then, flattening a tree into a list in the new representation can be done using the following function:

$$\begin{aligned} &\mathit{flatten}' :: \mathsf{Tree}\ \alpha \to \mathsf{DList}\ \alpha \\ &\mathit{flatten}'\ (\mathsf{Leaf}\ a) \quad\ = \mathit{consR}\ a\ \mathit{emptyR} \\ &\mathit{flatten}'\ (\mathsf{Node}\ t_1\ t_2) = \mathit{appendR}\ (\mathit{flatten}'\ t_1)\ (\mathit{flatten}'\ t_2) \end{aligned}$$

and a more efficient variant of the original function, with its original type, can be recovered as follows:

$$\begin{aligned} &\mathit{flatten} :: \mathsf{Tree}\ \alpha \to [\alpha] \\ &\mathit{flatten} = \mathit{abs} \circ \mathit{flatten}' \end{aligned}$$

There are two small problems with this approach. One is correctness. How do we know that the new *flatten* is equivalent to the original one? We could try to argue by "distributing" abs over the definition of $\mathit{flatten}'$, using $\mathit{abs}\ \mathit{emptyR} = [\,]$, $\mathit{abs}\ (\mathit{consR}\ a\ \mathit{as}) = a : \mathit{abs}\ \mathit{as}$, and

$$\mathit{abs}\ (\mathit{appendR}\ \mathit{as}\ \mathit{bs}) = \mathit{abs}\ \mathit{as} +\!\!+ \mathit{abs}\ \mathit{bs}\,. \tag{2}$$

But actually the last equation does not hold in general. The reason is that there are $\mathit{as} :: \mathsf{DList}\ \tau$ that are not in the image of rep. Consider, for example, $\mathit{as} = \mathsf{DL}\ \mathit{reverse}$. Then neither is $\mathit{as} = \mathit{rep}\ l$ for any l, nor does (2) hold for every bs. Any argument "by distributing abs" would thus have to rely on the implicit assumption that a certain discipline has been exercised when going from the original *flatten* to $\mathit{flatten}'$ by replacing $[\,]$, $(:)$, and $(+\!\!+)$ by emptyR, consR, and $\mathit{appendR}$ (and/or applying rep to explicit lists). But this implicit assumption is not immediately in reach for formal grasp. So it would be nice to be able to provide a single, conclusive correctness statement for transformations like the one above. One way to do so was presented by Voigtländer (2002), but it requires a certain restructuring of code that can hamper compositionality and flexibility by introducing abstraction at fixed program points (via lambda-abstraction and so-called vanish-combinators). This also brings us to the second problem with the simple approach above.

When, and how, should we switch between the original and the alternative representations of lists during program construction? If we first write the original version of *flatten* and only later, after observing a quadratic runtime overhead, switch manually to the $\mathit{flatten}'$-version, then this is quite cumbersome, in particular when it has to be done repeatedly for different functions. Of course, we could decide to always use emptyR, consR, and $\mathit{appendR}$ from the beginning, to be on the safe side. But actually this is not so safe, efficiency-wise, because the representation of lists by functions carries its own (constant-factor) overhead, and if a function does not use appends in a harmful way, then we do not want to pay this price. Using the alternative presentation in a particular situation should be a conscious decision, not a default. And assume that later on we change the behaviour of *flatten*, say, to explore only a single path through the input tree, so that no appends arise. Certainly, we do not want to have to go and manually switch back to the, now sufficient, original list representation. The cure to our woes here is obvious, and has often been applied in similar situations: simply use overloading. Specifically, we can declare a type constructor class as follows:

$$\begin{aligned} &\mathbf{class}\ \mathsf{ListLike}\ \delta\ \mathbf{where} \\ &\quad \mathit{empty} :: \delta\ \alpha \\ &\quad \mathit{cons} :: \alpha \to \delta\ \alpha \to \delta\ \alpha \\ &\quad \mathit{append} :: \delta\ \alpha \to \delta\ \alpha \to \delta\ \alpha \end{aligned}$$

and code *flatten* in the following form:

$$\begin{aligned} &\mathit{flatten} :: \mathsf{Tree}\ \alpha \to (\forall \delta.\ \mathsf{ListLike}\ \delta \Rightarrow \delta\ \alpha) \\ &\mathit{flatten}\ (\mathsf{Leaf}\ a) \quad\ = \mathit{cons}\ a\ \mathit{empty} \\ &\mathit{flatten}\ (\mathsf{Node}\ t_1\ t_2) = \mathit{append}\ (\mathit{flatten}\ t_1)\ (\mathit{flatten}\ t_2) \end{aligned}$$

Then, with the obvious instance definitions

$$\begin{aligned} &\mathbf{instance}\ \mathsf{ListLike}\ []\ \mathbf{where} \\ &\quad \mathit{empty} = [\,] \\ &\quad \mathit{cons} = (:) \\ &\quad \mathit{append} = (+\!\!+) \end{aligned}$$

and

$$\begin{aligned} &\mathbf{instance}\ \mathsf{ListLike}\ \mathsf{DList}\ \mathbf{where} \\ &\quad \mathit{empty} = \mathit{emptyR} \\ &\quad \mathit{cons} = \mathit{consR} \\ &\quad \mathit{append} = \mathit{appendR} \end{aligned}$$

we can use the single version of *flatten* above both to produce ordinary lists and to produce difference lists. The choice between the two will be made automatically by the type checker, depending on the context in which a call to *flatten* occurs. For example, in

$$\mathit{last}\ (\mathit{flatten}\ t) \tag{3}$$

the ordinary list representation will be used, due to the input type of last. Actually, (3) will compile (under GHC 6.6, at least) to exactly the same code as $\mathit{last}\ (\mathit{flatten}\ t)$ for the original definition of *flatten* from the very beginning of this section. Any overhead related to the type class abstraction is simply eliminated by a standard optimisation. In particular, this means that where the original representation of lists would have perfectly sufficed, programming against the abstract interface provided by the ListLike class does no harm either. On the other hand, (3) of course still suffers from the same quadratic runtime blow-up as with the original definition of *flatten*. But now we can switch to the better behaved difference list representation without touching the code of *flatten* at all, by simply using

$$\mathit{last}\ (\mathit{abs}\ (\mathit{flatten}\ t))\,. \tag{4}$$

Here the (input) type of abs determines *flatten* to use emptyR, consR, and $\mathit{appendR}$, leading to linear runtime.

Can we now also answer the correctness question more satisfactorily? Given the forms of (3) and (4), it is tempting to simply conjecture that $\mathit{abs}\ t = t$ for any t. But this cannot be quite right, as abs has different input and output types. Also, we have already

observed that some t of abs's input type are problematic by not corresponding to any actual list. The clou now is to only consider t that only use the ListLike interface, rather than any specific operations related to DList as such. That is, we will indeed prove that for every closed type τ and $t :: \mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$,

$$abs\ t_{\mathsf{DList}} = t_{[]}\,.$$

Since the polymorphism over δ in the type of t is so important, we follow Voigtländer (2008a) and make it an explicit requirement in a function that we will use instead of abs for switching from the original to the alternative representation of lists:

$$\begin{aligned} &improve :: (\forall\delta.\ \mathsf{ListLike}\ \delta \Rightarrow \delta\ \alpha) \to [\alpha] \\ &improve\ t = abs\ t \end{aligned}$$

Now, when we observe the problematic runtime overhead in (3), we can replace it by

$$last\ (improve\ (flatten\ t))\,.$$

That this does not change the semantics of the program is established by the following theorem, which provides the sought-after general correctness statement.

Theorem 6. *Let $t :: \mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$ for some closed type τ. Then*

$$improve\ t = t_{[]}\,.$$

Proof. We prove

$$\mathsf{unDL}\ t_{\mathsf{DList}}\ [\,] = t_{[]}\,, \tag{5}$$

which by the definitions of $improve$ and abs is equivalent to the claim. To do so, we first show that $\mathcal{F} : \mathsf{DList} \Leftrightarrow []$ with

$$\mathcal{F}\ \mathcal{R} = \mathsf{unDL}\ ;\ ([\mathcal{R}] \to [\mathcal{R}])\ ;\ (+\!\!+)^{-1}$$

is a ListLike-*action*, where the latter concept is defined as any relational action $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ for type constructors κ_1 and κ_2 that are instances of ListLike such that

- $(empty_{\kappa_1}, empty_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R}$,
- $(cons_{\kappa_1}, cons_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to (\mathcal{F}\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, and
- $(append_{\kappa_1}, append_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R} \to (\mathcal{F}\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$.

Indeed,

- $(emptyR, [\,]) \in \forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R}$, since for every $\mathcal{R}$ and $(l_1, l_2) \in [\mathcal{R}]$, $(\mathsf{unDL}\ emptyR\ l_1, [\,] +\!\!+ l_2) = (l_1, l_2) \in [\mathcal{R}]$,
- $(consR, (:)) \in \forall\mathcal{R}.\ \mathcal{R} \to (\mathcal{F}\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, since for every $\mathcal{R}$, $(a, b) \in \mathcal{R}$, $(f, bs) \in ([\mathcal{R}] \to [\mathcal{R}])\ ;\ (+\!\!+)^{-1}$, and $(l_1, l_2) \in [\mathcal{R}]$,

  $$\begin{aligned}&(\mathsf{unDL}\ (consR\ a\ (\mathsf{DL}\ f))\ l_1, (b : bs) +\!\!+ l_2) = \\ &(a : f\ l_1, b : bs +\!\!+ l_2) \in [\mathcal{R}]\end{aligned}$$

  by $(a, b) \in \mathcal{R}$ and $(f\ l_1, bs +\!\!+ l_2) \in [\mathcal{R}]$ (which holds due to $(f, (bs +\!\!+)) \in [\mathcal{R}] \to [\mathcal{R}]$ and $(l_1, l_2) \in [\mathcal{R}]$), and
- $(appendR, (+\!\!+)) \in \forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R} \to (\mathcal{F}\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, since for every $\mathcal{R}$, $(f, as) \in ([\mathcal{R}] \to [\mathcal{R}])\ ;\ (+\!\!+)^{-1}$, $(g, bs) \in ([\mathcal{R}] \to [\mathcal{R}])\ ;\ (+\!\!+)^{-1}$, and $(l_1, l_2) \in [\mathcal{R}]$,

  $$\begin{aligned}&(\mathsf{unDL}\ (appendR\ (\mathsf{DL}\ f)\ (\mathsf{DL}\ g))\ l_1, (as +\!\!+ bs) +\!\!+ l_2) = \\ &(f\ (g\ l_1), as +\!\!+ (bs +\!\!+ l_2)) \in [\mathcal{R}]\end{aligned}$$

  by $(f, (as +\!\!+)) \in [\mathcal{R}] \to [\mathcal{R}]$, $(g, (bs +\!\!+)) \in [\mathcal{R}] \to [\mathcal{R}]$, and $(l_1, l_2) \in [\mathcal{R}]$.

Hence, $(t_{\mathsf{DList}}, t_{[]}) \in \mathcal{F}\ id_\tau$. Given that we have $\mathcal{F}\ id_\tau = \mathsf{unDL}\ ;\ ([id_\tau] \to [id_\tau])\ ;\ (+\!\!+)^{-1} = \mathsf{unDL}\ ;\ (+\!\!+)^{-1}$, this implies $\mathsf{unDL}\ t_{\mathsf{DList}} = (t_{[]}\ +\!\!+)$, and thus (5).

Note that the ListLike-action $\mathcal{F} : \mathsf{DList} \Leftrightarrow []$ used in the above proof is the same as

$$\mathcal{F}\ \mathcal{R} = (\mathsf{DList}\ \mathcal{R})\ ;\ rep^{-1}\,,$$

given that $\mathsf{DList}\ \mathcal{R} = \mathsf{unDL}\ ;\ ([\mathcal{R}] \to [\mathcal{R}])\ ;\ \mathsf{DL}$. This connection suggests the following more general theorem, which can actually be proved much like above.

Theorem 7. *Let $t :: \mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$ for some closed type τ, let κ_1 and κ_2 be instances of* ListLike*, and let $h :: \kappa_2\ \alpha \to \kappa_1\ \alpha$. If*

- *$h\ empty_{\kappa_2} = empty_{\kappa_1}$,*
- *for every closed type τ, $a :: \tau$, and $as :: \kappa_2\ \tau$, $h\ (cons_{\kappa_2}\ a\ as) = cons_{\kappa_1}\ a\ (h\ as)$, and*
- *for every closed type τ and $as, bs :: \kappa_2\ \tau$, $h\ (append_{\kappa_2}\ as\ bs) = append_{\kappa_1}\ (h\ as)\ (h\ bs)$,*

then

$$h\ t_{\kappa_2} = t_{\kappa_1}\,.$$

Theorem 6 is a special case of this by setting $\kappa_1 = \mathsf{DList}$, $\kappa_2 = []$, and $h = rep$, and observing that

- $rep\ [\,] = emptyR$,
- for every closed type τ, $a :: \tau$, and $as :: [\tau]$, $rep\ (a : as) = consR\ a\ (rep\ as)$,
- for every closed type τ and $as, bs :: [\tau]$, $rep\ (as +\!\!+ bs) = appendR\ (rep\ as)\ (rep\ bs)$, and
- $abs \circ rep = id$,

all of which hold by easy calculations. One key observation here is that the third of the above observations does actually hold, in contrast to its faulty "dual" (2) considered earlier in this section.

Of course, free theorems can now also be derived for other types than those considered in Theorems 6 and 7. For example, for every closed type τ, $f :: \mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau \to \delta\ \tau$, and h as in Theorem 7, we get that:

$$f_{\kappa_1} \circ h = h \circ f_{\kappa_2}\,.$$

To apply the reasoning approach presented here to the generic setting of Voigtländer (2008a) would first require to extend it to *multi-parameter* type (constructor) classes, which are not currently part of the Haskell standard.

5. Discussion and Related Work

Of course, statements like that of Theorem 7 are not an entirely new revelation. That statement can be read as a typical fusion law for compatible morphisms between algebras over the signature described by the ListLike class declaration. (For a given τ, consider $\mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$ as the corresponding initial algebra, $\kappa_1\ \tau$ and $\kappa_2\ \tau$ as two further algebras, and the operation $\cdot_{\kappa_i}$ of instantiating a $t :: \mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$ to a $t_{\kappa_i} :: \kappa_i\ \tau$ as initial algebra morphism, or catamorphism. Then the conditions on h in Theorem 7 make it an algebra morphism and the theorem's conclusion, also expressible as $h \circ \cdot_{\kappa_2} = \cdot_{\kappa_1}$, is "just" that of the standard catamorphism fusion law.) But being able to derive such statements directly from the types in the language, based on its built-in abstraction facilities, immediately as well for more complicated types (like $\mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau \to \delta\ \tau$ instead of $\mathsf{ListLike}\ \delta \Rightarrow \delta\ \tau$), and all this without going through category-theoretic hoops, is new and unique to our approach (for higher-order polymorphism).

There has been quite some interest recently in enhancing the state of the art in reasoning about monadic programs. Filinski and

Støvring (2007) study induction principles for effectful data types. These principles are used for reasoning about functions on data types involving *specific* monadic effects (rather than about functions that are parametric over some monad), and based on the functions' *defining equations* (rather than based on their types only), and thus are orthogonal to our free theorems. But for their example applications to formal models of backtracking, Filinski and Støvring also use a form of relational reasoning very close to the one appearing in our invocation of relational parametricity. In particular, our Definition 1 corresponds to their Definition 3.3.[1] They also use monad morphisms (not to be confused with their monad-algebra morphisms, or rigid functions, playing the key role in their induction principles). The scope of their relational reasoning is different, though. They use it for establishing the observational equivalence of different implementations of the same monadic effect. This is, of course, one of the classical uses of relational parametricity: representation independence in different realizations of an abstract data type. But it is only *one* possible use, and our treatment of full polymorphism opens the door to other uses also in connection with monadic programs. Rather than only relating different, but semantically equivalent, implementations of the same monadic effect (as hard-wired into Filinski and Støvring's Definition 3.5), we actually connect monads embodying different effects. This leads to applications not previously in reach, such as our reasoning about preservation of invariants. It is worth pointing out that Filinski (2007) does use monad morphisms for "subeffecting", but only for the discussion of hierarchies inside each one of two competing implementations of the same set of monadic effects; the relational reasoning is then orthogonal to these hierarchies and again can only lead to statements about observational equivalence of the two realizations overall, rather than to more nuanced statements about programs in one of them as such.

Swierstra (200x) proposes to code against modularly assembled *free* monads, where the assembling takes place by building coproducts of signature functors corresponding to the term languages of free monads. The associated type signatures are able to convey some of the information captured by our approach. For example, a monadic type $\mathsf{Term\ PutStr\ Int}$ can be used to describe computations whose only possible side-effect is that of writing strings to the output. Passing a list of values of that type to a function $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ clearly results in a value of type $\mathsf{Term\ PutStr\ Int}$ as well. Thus, if it is guaranteed (note the proof obligation) that "execution" of such a term value, on a kind of virtual machine (Swierstra and Altenkirch 2007) or in the actual IO monad, does indeed have no other side effect than potential output, then one gets a statement in the spirit of our Example 6. On the other hand, statements like the one in our Example 8 (also, say, reformulated for exceptions in the IO monad) are not in reach with that approach alone. Moreover, Swierstra's approach to "subeffecting" depends very much on syntax, essentially on term language inclusion along with proof obligations on the execution functions from terms to some semantic space. This would make obtaining statements roughly analogous to our Examples 5 and 7 extremely cumbersome. And ultimately, depending on syntactic inclusion is too strong a restriction in any case. For example, $putStr$ "" is semantically equivalent to $return\ ()$, and thus without visible side-effect. But nevertheless, any computation syntactically containing a call to $putStr$ would of necessity be assigned a type in a monad $\mathsf{Term}\ g$ with g "containing" (with respect to Swierstra's functor-level relation $:\prec:$) the functor PutStr, even when that call's argument would eventually evaluate to the empty string. Thus, such a computation would be banned from the input list in a statement like the one we give below Example 6. It is not so with our more semantical approach.

[1] It is unclear whether Filinski and Støvring are aware that it can be traced back to the treatment of a type class constraint by Wadler (1989, Section 3.4).

References

L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Principles of Programming Languages, Proceedings*, pages 284–294. ACM Press, 1996.

A. Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375(1–3):41–75, 2007.

A. Filinski and K. Støvring. Inductive reasoning about effectful data types. In *International Conference on Functional Programming, Proceedings*, pages 97–110. ACM Press, 2007.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

R.J.M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

J. Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and "Free Theorems"*. PhD thesis, Massachusetts Institute of Technology, 1997.

J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages, Proceedings*, pages 333–343. ACM Press, 1995.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. Submitted to *International Conference on Functional Programming*, 2008.

W. Swierstra. Data types à la carte. *Journal of Functional Programming*, to appear, 200x.

W. Swierstra and T. Altenkirch. Beauty in the beast — A functional semantics for the awkward squad. In *Haskell Workshop, Proceedings*, pages 25–36. ACM Press, 2007.

I. Takeuti. The theory of parametricity in lambda cube. Draft, 2001.

J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.

J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, LNCS. Springer-Verlag, 2008a. To appear.

J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008b.

D. Vytiniotis and S. Weirich. Type-safe cast does no harm. Draft, 2007.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76, 1989.