Free Theorems Involving Type Constructor Classes

Functional Pearl

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany
janis.voigtlaender@acm.org

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

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 strength should hold for reasoning about programs seem not to be realised 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 observation pertains to functions from the Prelude (Haskell's standard library) like

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

but also to many user-defined functions. Such abstraction 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:[1]

$$\begin{aligned}
f_1 &= head \\
f_2\ ms &= sequence\ ms \gg\!= return \circ sum \\
f_3 &= f_2 \circ reverse \\
f_4\ [\,] &= return\ 0 \\
f_4\ (m : ms) &= \mathbf{do}\ \ i \leftarrow m \\
&\qquad\qquad \mathbf{let}\ l = length\ ms \\
&\qquad\qquad \mathbf{if}\ i > l\ \mathbf{then}\ return\ (i + l) \\
&\qquad\qquad\qquad \mathbf{else}\ f_4\ (drop\ i\ ms)
\end{aligned}$$

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 forth. 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 limitation 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 statement is quite intuitive from the above observation about f being unable to cause new effects on

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

[1] The functions *head*, *sum*, *reverse*, *length*, and *drop* are all from the Prelude. Their general types and explanation can be found via Hoogle (`http://haskell.org/hoogle`). The notation ∘ is for function composition, while ≫= and **do** are two different syntaxes for performing computations in a monad one after the other. Finally, *return* embeds pure values into a monad.

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 simplification 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 that applies 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), to specific applications in software engineering and databases (Voigtländer 2009).

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. Since we are mostly interested in demonstrating the prospects gained from that combination, we refrain here from developing the folklore into a full-fledged formal apparatus that would stand to blur the intuitive ideas. This is not an overly theoretical paper. Also on purpose, we do not consider Haskell intricacies, like those studied by Johann and Voigtländer (2004) and Stenger and Voigtländer (2009), 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 (but consider the extension to general recursion in Appendix C). For the sake of accessibility, we also stay close to Wadler's notation.

2. Free Theorems, in Full Beauty

So what is the deal with free theorems? Why should it be possible to derive statements about a function's behaviour from its type alone? Maybe it is best to start with a concrete example. Consider the type signature

$$f :: [\alpha] \to [\alpha]\,.$$

What does it tell us about the function f? For sure that it takes lists as input and produces lists as output. But we also see that f is polymorphic, due to the type variable α, and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list can only ever contain elements from the input list. For the function, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, or even id, because then f would immediately fail to have the general type $[\alpha] \to [\alpha]$.[2]

> So for any input list l (over any element type) the output list $f\ l$ consists solely of elements from l.

But how can f decide which elements from l to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list l. For in a pure functional language f has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in any way about what to do. And the means by which to make decisions based on l are limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of l. For the function is ignorant of the element type, and so is prevented from analysing list elements in any way (be it by pattern-matching, comparison operations, or whatever). In fact, the only means for f to drive its decision-making is to inspect the *length* of l, because that is the only element-independent "information content" of a list.

> So for any pair of lists l and l' of same length (but possibly over different element types) the lists $f\ l$ and $f\ l'$ are formed by making the same position-wise selections of elements from l and l', respectively.

Now consider the following standard Haskell function:

$$\begin{array}{l} map :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ map\ g\ [\,] \qquad\ = [\,] \\ map\ g\ (a : as) = (g\ a) : (map\ g\ as) \end{array}$$

Clearly, $map\ g$ for any g preserves the lengths of lists. So if $l' = map\ g\ l$, then $f\ l$ and $f\ l'$ are of the same length and contain, at each position, position-wise exactly corresponding elements from l and l', respectively. Since, moreover, any two position-wise corresponding elements, one from l and one from $l' = map\ g\ l$, are related by the latter being the g-image of the former, we have that at each position $f\ l'$ contains the g-image of the element at the same position in $f\ l$.

> So for any list l and (type-appropriate) function g, we have $f\ (map\ g\ l) = map\ g\ (f\ l)$.

Note that during the reasoning leading up to that statement we did not (need to) consider the actual definition of f at all. The methodology of deriving free theorems à la Wadler (1989) is a way to obtain statements of this flavour for arbitrary function types, and in a more disciplined (and provably sound) manner than the mere handwaving performed above.

The key to doing so is to interpret types as relations. For example, given the 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

[2] The situation is more complicated in the presence of general recursion. For further discussion, see Appendix C.

- 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, $u_{\tau_1} :: \tau[\tau_1/\alpha]$ 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 (the least fixpoint of)

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

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 insight 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 $\tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, $l :: [\tau_1]$, and $l' :: [\tau_2]$ that $(l, l') \in [\mathcal{R}]$ implies $(f\ l, f\ l') \in [\mathcal{R}]$, or, specialised to the function level ($\mathcal{R} \mapsto g$, and thus $[\mathcal{R}] \mapsto \mathit{map}\ g$), for every $g :: \tau_1 \to \tau_2$ and $l :: [\tau_1]$ that $f\ (\mathit{map}\ g\ l) = \mathit{map}\ g\ (f\ l)$. This proof finally provides the formal counterpart to the intuitive reasoning earlier in this section. And the development is algorithmic enough that it can be performed automatically. Indeed, an online free theorems generator (Böhme 2007) is accessible at our homepage (`http://linux.tcs.inf.tu-dresden.de/~voigt/ft/`).

3. The Extension to Type Constructor Classes

We now want 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 seldom 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 above. 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 (2009).

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 (seen as a new constant in the language) is related to itself by the relational interpretation of its type. This relatedness then guarantees that the overall result (i.e., that the relational interpretation of every closed type is an identity relation) stays intact (Mitchell and Meyer 1985). The same approach immediately applies to type constructor classes as well. Consider, for example, the Monad class declaration:

$$\begin{aligned}&\mathbf{class}\ \mathsf{Monad}\ \mu\ \mathbf{where}\\ &\quad \mathit{return} :: \alpha \to \mu\ \alpha\\ &\quad (\gg\!=) :: \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta\end{aligned}$$

Since the type of return is $\forall\mu.\ \mathsf{Monad}\ \mu \Rightarrow (\forall\alpha.\ \alpha \to \mu\ \alpha)$, we expect that $(\mathit{return}, \mathit{return}) \in \forall\mathcal{F}.\ \mathsf{Monad}\ \mathcal{F} \Rightarrow (\forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\ \mathcal{R})$, 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

- $(\mathit{return}_{\kappa_1}, \mathit{return}_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\,\mathcal{R}$ 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*.[3] (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:

$$\begin{aligned}&\mathbf{instance}\ \mathsf{Monad}\ \mathsf{Maybe}\ \mathbf{where}\\ &\quad \mathit{return}\ a = \mathsf{Just}\ a\\ &\quad \mathsf{Nothing} \gg\!= k = \mathsf{Nothing}\\ &\quad \mathsf{Just}\ a \quad \gg\!= k = k\ a\end{aligned}$$

$$\begin{aligned}&\mathbf{instance}\ \mathsf{Monad}\ []\ \mathbf{where}\\ &\quad \mathit{return}\ a = [a]\\ &\quad \mathit{as} \gg\!= k = \mathit{concat}\ (\mathit{map}\ k\ \mathit{as})\end{aligned}$$

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

$$\begin{aligned}&\mathcal{R} = \mathcal{S} = \mathit{id}_{\mathsf{Int}}\,,\\ &m_1 = \mathsf{Just}\ 1\,,\\ &m_2 = [1, 2]\,,\\ &k_1 = \lambda i \to \mathbf{if}\ i > 1\ \mathbf{then}\ \mathsf{Just}\ i\ \mathbf{else}\ \mathsf{Nothing}\,,\ \text{and}\\ &k_2 = \lambda i \to \mathit{reverse}\ [2..i]\,.\end{aligned}$$

Clearly, $(m_1, m_2) \in \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$ and $(k_1, k_2) \in \mathit{id}_{\mathsf{Int}} \to \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$, but $(m_1 \gg\!=_{\mathsf{Maybe}} k_1, m_2 \gg\!=_{[]} k_2) = (\mathsf{Nothing}, [2]) \notin \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$. On the other hand, the relational action $\mathcal{F}' : \mathsf{Maybe} \Leftrightarrow []$ with

$$\mathcal{F}'\,\mathcal{R} = \{(\mathsf{Nothing}, [\,])\} \cup \{(\mathsf{Just}\ a, [b]) \mid (a, b) \in \mathcal{R}\}$$

is a Monad-action.

[3] It is worth noting that "dictionary translation" (Wadler and Blott 1989) would be an alternative way of motivating this definition.

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 of examples of interesting results obtained for concrete monads.

An important role will be played by a notion connecting different Monad instances on a functional, rather than relational, level.

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

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

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

then h is called a Monad-*morphism*.

The two notions of Monad-action and Monad-morphism are strongly related, in that Monad-actions are closed under pointwise composition with Monad-morphisms or the inverses thereof, depending on whether the composition is from the left or from the right (Filinski and Støvring 2007, Proposition 3.7(2)).

4. One Application Field: 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 function types involving quantification over Monad-restricted type constructor variables just as well. And note that the presence of the concrete type Int in the function signature makes any results we obtain for such f more, rather than less, interesting. For clearly there are strictly, and considerably, fewer functions of type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \to \mu\ \alpha$ than there are of type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ [4], so proving a statement for all functions of the latter type demonstrates much more power than proving the same statement for all functions of the former type only. In other words, telling f what type of values are encapsulated in its monadic inputs and output entails more possible behaviours of f that our reasoning principle has to keep under control.

Also, it is *not* the case that using Int in most examples in this section means that we might as well have monomorphised the monad interface as follows:

$$\begin{aligned}&\mathbf{class}\ \mathsf{IntMonad}\ \mu\ \mathbf{where}\\ &\quad return :: \mathsf{Int} \to \mu\\ &\quad (\gg\!=) :: \mu \to (\mathsf{Int} \to \mu) \to \mu\end{aligned}$$

and thus are actually just proving results about a less interesting type $\mathsf{IntMonad}\ \mu \Rightarrow [\mu] \to \mu$ without any higher-orderedness (viz., quantifying only over a type variable rather than over a type *constructor* variable). This impression would be a misconception, as we *do* indeed prove results for functions critically depending on the use of higher-order polymorphism. That the type under consideration is $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ does by no way mean that monadic encapsulation is restricted to only integer values inside functions of that type. Just consider the function f_2 from the introduction. During that function's computation, the monadic bind operation $(\gg\!=)$ is used to combine a μ-encapsulated integer list (viz., $sequence\ ms :: \mu\ [\mathsf{Int}]$) with a function to a μ-encapsulated single integer (viz., $return \circ sum :: [\mathsf{Int}] \to \mu\ \mathsf{Int}$). Clearly, the same or similarly modular code could not have been written at type $f_2 :: \mathsf{IntMonad}\ \mu \Rightarrow [\mu] \to \mu$, because there is no way to provide a function like $sequence$ for the IntMonad class (or any single monomorphised class), not even when we are content with making $sequence$ less flexible by fixing the α in its current type to be Int. So again, proving results about all functions of type $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ covers more ground than might at first appear to be the case.

Having rationalised our choice of example function type, let us now get some actual work done. As a final preparation, we need to mention three laws that Monad instances κ are often expected to satisfy:[5]

$$return_\kappa\ a \gg\!=_\kappa k\ =\ k\ a \tag{1}$$

$$m \gg\!=_\kappa return_\kappa\ =\ m \tag{2}$$

$$(m \gg\!=_\kappa k) \gg\!=_\kappa q\ =\ m \gg\!=_\kappa (\lambda a \to k\ a \gg\!=_\kappa q) \tag{3}$$

Since Haskell does not enforce these laws, and it is easy to define Monad instances violating them, we will explicitly keep track of where the laws are required in our statements and proofs.

4.1 Purity Preservation

As mentioned in the introduction, one first intuitive statement we naturally expect to hold of any $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ 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 as follows. It is proved in Appendix A, and is a corollary of Theorem 3 (to be given later).

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

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

[4] After all, any function (definition) of the type polymorphic over α can also be given the more specific type, whereas of the functions f_1 to f_4 given in the introduction as examples for functions of the latter type only f_1 can be given the former type as well.

[5] Indeed, only a Monad instance satisfying these laws constitutes a "monad" in the mathematical sense of the word.

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 propagation 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 monad law (1).

4.2 Safe Value Extraction

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 in the spirit of $unsafePerformIO :: \mathsf{IO}\ \alpha \to \alpha$, 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 reduction to work for arbitrary "projections", or even arbitrary monads. Rather, we need to devise appropriate restrictions and prove that they suffice. The formal statement is 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* $g :: [\mathsf{Int}] \to \mathsf{Int}$*.*[6]

The theorem is proved in Appendix B. Also, it is a corollary of Theorem 4. Note that no monad laws at all are needed in Theorem 2 and its proof. The same will be true for the other theorems we are going to provide, 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):

$$\begin{array}{l}\mathbf{newtype}\ \mathsf{Writer}\ \alpha = \mathsf{Writer}\ (\alpha, \mathsf{String})\\[1ex]\mathbf{instance}\ \mathsf{Monad}\ \mathsf{Writer}\ \mathbf{where}\\ \quad return\ a = \mathsf{Writer}\ (a, \text{``''})\\ \quad \mathsf{Writer}\ (a, s) \gg\!= k =\\ \quad\quad \mathsf{Writer}\ (\mathbf{case}\ k\ a\ \mathbf{of}\ \mathsf{Writer}\ (a', s') \to (a', s +\!\!+ s'))\end{array}$$

[6] In fact, this g is explicitly given as follows: $\mathsf{unId} \circ f_{\mathsf{Id}} \circ (map\ \mathsf{Id})$, using the type constructor Id and its Monad instance definition from Appendix A.

Assume we are interested in applying an $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$ to an $l :: [\mathsf{Writer}\ \mathsf{Int}]$, 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:

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

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 $l' :: [\mathsf{Writer}\ \mathsf{Int}]$ encapsulating the same integer values, but 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 Section 3. 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 factorisation fails in a subtle way. Consider, for example, the (for the sake of simplicity, artificial) function

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

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 3. 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.)

4.3 Monad Subspacing

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 realise 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 realise 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 realise 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 preservation 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 ($\mathsf{Either\ String}$), 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 Definition 2 and the following theorem, as well as in the subsequent examples.

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

Proof. We prove that for every $l' :: [\kappa_1\ \mathsf{Int}]$,

$$f_{\kappa_2}\ (map\ h\ l') = h\ (f_{\kappa_1}\ l')\,. \tag{4}$$

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

$$\mathcal{F}\ \mathcal{R} = (\kappa_2\ \mathcal{R})\ ;\ h^{-1}\,,$$

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

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

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

  by $((\gg\!=_{\kappa_2}), (\gg\!=_{\kappa_2})) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \kappa_2\ \mathcal{R} \to ((\mathcal{R} \to \kappa_2\ \mathcal{S}) \to \kappa_2\ \mathcal{S})$, $(m_1, h\ m_2) \in \kappa_2\ \mathcal{R}$, and $(k_1, h \circ k_2) \in \mathcal{R} \to \kappa_2\ \mathcal{S}$.

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

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

Example 5. Consider the well-known reader monad:

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

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 it does hold by Theorem 3 with the following Monad-morphism:

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

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:

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

What is left to do is to show that h is a Monad-morphism. But this property follows from $putStr\ "" = return\ ()$,

$putStr\ (s +\!\!+ s') = putStr\ s \gg putStr\ s'$, and monad laws (1) and (3) 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:

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

Intuitively, this monad 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:

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

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).

4.4 Effect Abstraction

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* $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$ *be a* Monad*-morphism. Then* $h \circ f_{\kappa_1}$ *gives the same result for any two lists of same length whose corresponding elements have the same* h*-images.*

Proof. Let $l_1, l_2 :: [\kappa_1\ \mathsf{Int}]$ be such that $map\ h\ l_1 = map\ h\ l_2$. Then $h\ (f_{\kappa_1}\ l_1) = h\ (f_{\kappa_1}\ l_2)$ by statement (4) 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 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 statement holds by Theorem 4 with the following Monad-morphism:

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

4.5 A More Polymorphic Example

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* from the introduction, also showing that mixed quantification over both type constructor variables and ordinary type variables can very well be handled. The theorem's statement involves the following function:

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

Theorem 5. *Let* $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \to \mu\ [\alpha]$ *and let* $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$ *be a* Monad*-morphism. If* κ_2 *satisfies law (2), then for every choice of closed types* τ_1 *and* τ_2 *and* $g :: \tau_1 \to \tau_2$*,*

$$f_{\kappa_2} \circ map\ (fmap_{\kappa_2}\ g) \circ map\ h = fmap_{\kappa_2}\ (map\ g) \circ h \circ f_{\kappa_1}.$$[7]

Intuitively, this theorem means that any f of type $\mathsf{Monad}\ \mu \Rightarrow [\mu\ \alpha] \to \mu\ [\alpha]$ commutes with, both, transformations on the monad structure and transformations on the element level. The occurrences of map and $fmap$ are solely there to bring those transformations h and g into the proper positions with respect to the different nestings of the type constructors μ and [] on the input and output sides of f. Note that by setting either g or h to id, we obtain the specialised versions

$$f_{\kappa_2} \circ map\ h = h \circ f_{\kappa_1}$$

[7] For the curious reader: the proof derives this statement from $(f_{\kappa_2}, f_{\kappa_1}) \in [\mathcal{F}\ g^{-1}] \to \mathcal{F}\ [g^{-1}]$ for the same Monad-action $\mathcal{F} : \kappa_2 \Leftrightarrow \kappa_1$ as used in the proof of Theorem 3.

and

$$f_\kappa \circ map\ (fmap_\kappa\ g) = fmap_\kappa\ (map\ g) \circ f_\kappa\,. \qquad (5)$$

Further specialising the latter by choosing the identity monad for κ, we would also essentially recover the free theorem derived for $f :: [\alpha] \to [\alpha]$ in Section 2.

5. 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 so for flattening a tree of type

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

using the following function:

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

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:

$$\mathbf{newtype}\ \mathsf{DList}\ \alpha = \mathsf{DL}\ \{\mathsf{unDL} :: [\alpha] \to [\alpha]\}$$

$$\begin{array}{l} \mathit{rep} :: [\alpha] \to \mathsf{DList}\ \alpha \\ \mathit{rep}\ l = \mathsf{DL}\ (l +\!\!+) \end{array}$$

$$\begin{array}{l} \mathit{abs} :: \mathsf{DList}\ \alpha \to [\alpha] \\ \mathit{abs}\ (\mathsf{DL}\ f) = f\ [\,] \end{array}$$

$$\begin{array}{l} \mathit{emptyR} :: \mathsf{DList}\ \alpha \\ \mathit{emptyR} = \mathsf{DL}\ \mathit{id} \end{array}$$

$$\begin{array}{l} \mathit{consR} :: \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\ \mathit{consR}\ a\ (\mathsf{DL}\ f) = \mathsf{DL}\ ((a :) \circ f) \end{array}$$

$$\begin{array}{l} \mathit{appendR} :: \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\ \mathit{appendR}\ (\mathsf{DL}\ f)\ (\mathsf{DL}\ g) = \mathsf{DL}\ (f \circ g) \end{array}$$

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

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

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

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

There are two 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 $abs\ \mathit{emptyR} = [\,]$, $abs\ (\mathit{consR}\ a\ as) = a : abs\ as$, and

$$abs\ (\mathit{appendR}\ as\ bs) = abs\ as +\!\!+ abs\ bs\,. \qquad (6)$$

But actually the last equation does not hold in general. The reason is that there are $as :: \mathsf{DList}\ \tau$ that are not in the image of rep. Consider, for example, $as = \mathsf{DL}\ \mathit{reverse}$. Then neither is $as = \mathit{rep}\ l$ for any l, nor does (6) 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 rewriting 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 strategy is not so safe, efficiency-wise, because the representation of lists by functions carries its own (constant-factor) overhead. If a function does not use appends in a harmful way, then we do not want to pay this price. Hence, 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 at all 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 almost obvious, and has often been applied in similar situations: simply use overloading. Specifically, we can declare a type constructor class as follows:

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

and code *flatten* in the following form:

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

Then, with the obvious instance definitions

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

and

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

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

$$last\ (\mathit{flatten}\ t) \qquad (7)$$

the ordinary list representation will be used, due to the input type of $last$. Actually, (7) will compile (under GHC, at least) to exactly the same code as $last\ (\mathit{flatten}\ t)$ 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, (7) 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

$$last\ (abs\ (flatten\ t))\,. \tag{8}$$

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

Can we now also answer the correctness question more satisfactorily? Given the forms of (7) and (8), it is tempting to simply conjecture that $abs\ t = t$ for any t. But this conjecture 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 coup 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 (7), we can replace it by

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

That this replacement 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

$$\mathsf{unDL}\ t_{\mathsf{DList}} = (t_{[]}\ +\!\!+)\,, \tag{9}$$

which by the definitions of *improve* and abs, and by $t_{[]}\ +\!\!+\ [\,] = t_{[]}$, implies 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 (9).

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 $h :: \kappa_1\ \alpha \to \kappa_2\ \alpha$. If*

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

then

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

Theorem 6 is a special case of this theorem by setting $\kappa_1 = []$, $\kappa_2 = \mathsf{DList}$, 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" (6) 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:

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

6. 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 $h \circ \cdot_{\kappa_1} = \cdot_{\kappa_2}$, 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.

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. 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. These connections lead 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 (via Monad-actions and so forth) 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. The reason again, as with Filinski and Støvring (2007), is that no full polymorphism is considered, but only parametrisation over same-effect-monads on top-level. Interestingly, though, the key step in all our proofs in Section 4, namely finding a suitable Monad-action, can be streamlined in the spirit of Proposition 3.7 of Filinski and Støvring (2007) or Lemmas 45, 46 of Filinski (2007). It seems fair to mention that the formal accounts of Filinski and Støvring are very complex, but that this is necessarily so because they deal with general recursion at both term and type level, while we have completely dodged such issues. Treating general recursion in a semantic framework typically involves a good deal of domain theory such as considered by Birkedal et al. (2007). We only provide a very brief sketch of what interactions we expect between general recursion and our developments from the previous sections in Appendix C.

Swierstra (2008) 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 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 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 dependence prevents directly obtaining statements roughly analogous to our Examples 5 and 7 using his approach. Also, depending on syntactic inclusion is a very strong restriction indeed. 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 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.

Dealing more specifically with concrete monads is the topic of recent works by Hutton and Fulger (2008), using point-free equational reasoning, and by Nanevski et al. (2008), employing an axiomatic extension of dependent type theory.

On the tool side, we already mentioned the free theorems generator at `http://linux.tcs.inf.tu-dresden.de/~voigt/ft/`. It deals gracefully with ordinary type classes (in the offline, shell-based version even with user-defined ones), but has not yet been extended for type *constructor* classes. There is also another free theorems generator, written by Andrew Bromage, running in Lambdabot (`http://haskell.org/haskellwiki/Lambdabot`). It does not know about type or type constructor *classes*, but deals with type constructors by treating them as fixed functors. Thus, it can, for example, derive the statement (5) for functions $f_\kappa :: [\kappa\ \alpha] \to \kappa\ [\alpha]$, but not more general and more interesting statements like those given in Theorem 5 and earlier, connecting different Monad instances, concerning the beyond-functor aspects of monads, or our results about ListLike.

Acknowledgments

I would like to thank the anonymous reviewers of more than one version of this paper who have helped to improve it through their criticism and suggestions. Also, I would like to thank Helmut Seidl, who inspired me to consider free theorems involving type constructor classes in the first place by asking a challenging question regarding the power of type(-only)-based reasoning about monadic programs during a train trip through Munich quite some time ago. (The answer to his question is essentially Example 7.)

References

L. Birkedal, R.E. Møgelberg, and R.L. Petersen. Domain-theoretical models of parametric polymorphism. *Theoretical Computer Science*, 388 (1–3):152–172, 2007.

S. Böhme. Free theorems for sublanguages of Haskell. Master's thesis, Technische Universität Dresden, 2007.

N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

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.

G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming, Draft Proceedings*, 2008.

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.

J.C. Mitchell and A.R. Meyer. Second-order logical relations (Extended abstract). In *Logic of Programs, Proceedings*, volume 193 of *LNCS*, pages 225–236. Springer-Verlag, 1985.

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

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming, Proceedings*, pages 229–240. ACM Press, 2008.

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. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.

W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

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. Manuscript, 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*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008a.

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

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

D. Vytiniotis and S. Weirich. Type-safe cast does no harm: Syntactic parametricity for F_ω and beyond. Manuscript, 2009.

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. ACM Press, 1989.

A. Proof of Theorem 1

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

$$\begin{array}{l}\mathbf{newtype}\ \mathsf{Id}\ \alpha = \mathsf{Id}\ \{\mathsf{unId} :: \alpha\}\\[2ex]\mathbf{instance}\ \mathsf{Monad}\ \mathsf{Id}\ \mathbf{where}\\ \quad return\ a = \mathsf{Id}\ a\\ \quad \mathsf{Id}\ a \gg\!= k = k\ a\end{array}$$

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 (forward) 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 monad law (1) for κ.)

Hence, by what we derived towards the end of Section 3, $(f_\kappa, f_{\mathsf{Id}}) \in [\mathcal{F}\ id_{\mathsf{Int}}] \to \mathcal{F}\ id_{\mathsf{Int}}$. Given that we have $\mathcal{F}\ id_{\mathsf{Int}} = return_\kappa^{-1}\ ;\ \mathsf{Id} = (return_\kappa \circ \mathsf{unId})^{-1}$, this implies the claim.

B. Proof of Theorem 2

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 $(k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$, $(m \gg\!=_\kappa\ k_1, \mathsf{Id}\ b \gg\!=_{\mathsf{Id}}\ k_2) \in p\ ;\ \mathcal{S}\ ;\ \mathsf{Id}$ by $p\ (m \gg\!=_\kappa\ k_1) = p\ (k_1\ (p\ m))$ and $(k_1\ (p\ m), k_2\ b) \in p\ ;\ \mathcal{S}\ ;\ \mathsf{Id}$ (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.

C. Free Theorems, the Ugly Truth

Free theorems as described in Section 2 are beautiful. And very nice. Almost too good to be true. And actually they are not. At least not unrestricted and in a setting more closely resembling a modern

functional language than the plain polymorphic lambda-calculus for which relational parametricity was originally conceived. In particular, problems are caused by general recursion with its potential for nontermination. We have purposefully ignored this issue throughout the main body of the paper, so as to be able to explain our ideas and new abstractions in the most basic surrounding. In a sense, our reasoning has been "up to $\bot$", or "fast and loose, but morally correct" (Danielsson et al. 2006). We leave a full formal treatment of free theorems involving type constructor classes in the presence of partiality as a challenge for future work, but use this appendix to outline some refinements that are expected to play a central role in such a formalisation.

So what is the problem with potential nontermination? Let us first discuss this question based on the simple example

$$f :: [\alpha] \to [\alpha]$$

from Section 2. There, we argued that the output list of any such f can only ever contain elements from the input list. But this claim is not true anymore now, because f might just as well choose, for some element position of its output list, to start an arbitrary looping computation. That is, while f certainly (and still) cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, it may very well put $\bot$ there, even while not knowing the element type of the lists it operates over, because $\bot$ does exist at every type. So the erstwhile claim that for any input list l the output list $f\ l$ consists solely of elements from l has to be refined as follows.

> For any input list l the (potentially partial or infinite) output list $f\ l$ consists solely of elements from l and/or $\bot$.

The decisions about which elements from l to propagate to the output list, in which order and multiplicity, and where to put $\bot$ can again only be made based on the input list l, and only by inspecting its length (or running into an undefined tail or an infinite list).

> So for any pair of lists l and l' of same length (refining this notion to take partial and infinite lists into account) the lists $f\ l$ and $f\ l'$ are formed by making the same position-wise selections of elements from l and l', respectively, and by inserting $\bot$ at the same positions, if any.

For any $l' = \mathit{map}\ g\ l$, we then still have that $f\ l$ and $f\ l'$ are of the same length and contain position-wise exactly corresponding elements from l and $l' = \mathit{map}\ g\ l$, *at those positions where f takes over elements from its input rather than inserting* $\bot$. For those positions where f *does* insert $\bot$, which will then happen equally for $f\ l$ and $f\ l'$, we may only argue that the element in $f\ l'$ contains the g-image of the corresponding element in $f\ l$ if indeed $\bot$ is the g-image of $\bot$, that is, if g is a strict function.

> So for any list l and, importantly, *strict* function g, we have $f\ (\mathit{map}\ g\ l) = \mathit{map}\ g\ (f\ l)$.

The formal counterpart to the extra care exercised above regarding potential occurrences of $\bot$ is the provision of Wadler (1989, Section 7) that only *strict and continuous relations* should be allowed as interpretations for types.

In particular, when interpreting quantification over type variables by quantification over relation variables, those quantified relations are required to contain the pair $(\bot, \bot)$, also signified via the added $\cdot$ in the new notation $\mathcal{R} : \tau_1 \mathrel{\dot{\Leftrightarrow}} \tau_2$. With straightforward changes to the required constructions on relations, such as explicitly including the pair $(\bot, \bot)$ in $[\mathcal{R}] : [\tau_1] \mathrel{\dot{\Leftrightarrow}} [\tau_2]$ and $\mathsf{Maybe}\ \mathcal{R} : \mathsf{Maybe}\ \tau_1 \mathrel{\dot{\Leftrightarrow}} \mathsf{Maybe}\ \tau_2$, and replacing the least by the greatest fixpoint in the definition of $[\mathcal{R}]$, we get a treatment of free theorems that is sound even for a language including general recursion, and thus nontermination.

For the extension to the setting with type constructor classes (cf. Section 3), we will need to mandate that any relational action, now denoted $\mathcal{F} : \kappa_1 \mathrel{\dot{\Leftrightarrow}} \kappa_2$, must preserve strictness, i.e., map $\mathcal{R} : \tau_1 \mathrel{\dot{\Leftrightarrow}} \tau_2$ to $\mathcal{F}\ \mathcal{R} : \kappa_1\ \tau_1 \mathrel{\dot{\Leftrightarrow}} \kappa_2\ \tau_2$. Apart from that, Definition 1, for example, is expected to remain unchanged (except that $\mathcal{R}$ and $\mathcal{S}$ will now range over strict relations, of course).

Under these assumptions, we can investigate the impact of the presence of general recursion on the results seen in the main body of this paper. Consider Theorem 1, for example. In order to have $\mathcal{F} : \kappa \mathrel{\dot{\Leftrightarrow}} \mathsf{Id}$ in its proof, we need to change the definition of $\mathcal{F}\ \mathcal{R}$ as follows:

$$\mathcal{F}\ \mathcal{R} = \{(\bot, \bot)\} \cup (\mathit{return}_\kappa^{-1}\ ;\ \mathcal{R}\ ;\ \mathsf{Id})\,.$$

For this relational action to be a Monad-action, we would need the additional condition that $\bot \gg\!=_\kappa k_1 = k_1\ \bot$ for any choice of k_1. Then, $(f_\kappa, f_{\mathsf{Id}}) \in [\mathcal{F}\ \mathit{id}_{\mathsf{Int}}] \to \mathcal{F}\ \mathit{id}_{\mathsf{Int}}$ would allow to derive the following variant, valid in the presence of general recursion and $\bot$.

> **Theorem 1'.** *Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, let κ be an instance of Monad satisfying law (1) and $\bot \gg\!=_\kappa k = k\ \bot$ for every (type-appropriate) k, and let $l :: [\kappa\ \mathsf{Int}]$. If every element in l is a return_κ-image or $\bot$, then so is $f_\kappa\ l$.*

Note that the Reader monad, for example, satisfies the conditions for applying the thus adapted theorem.

Similar repairs are conceivable for the other statements we have derived, or one might want to derive. Just as another sample, we expect Example 7 to change as follows.

> **Example 7'.** Let $f :: \mathsf{Monad}\ \mu \Rightarrow [\mu\ \mathsf{Int}] \to \mu\ \mathsf{Int}$, let τ be a closed type, and let $l :: [\mathsf{State}\ \tau\ \mathsf{Int}]$. If for every element $\mathsf{State}\ g$ in l, the property $P(g)$ defined as
>
> $$P(g)\ :=\ \forall s.\ \mathit{snd}\ (g\ s) = s \vee \mathit{snd}\ (g\ s) = \bot$$
>
> holds, then also $f\ l$ is of the form $\mathsf{State}\ g$ for some g with $P(g)$.

Note that even if we had kept the stronger precondition that $\mathit{snd} \circ g = \mathit{id}$ for every element $\mathsf{State}\ g$ in l, it would be impossible to prove $\mathit{snd} \circ g = \mathit{id}$ instead of the weaker $P(g)$ for $f\ l = \mathsf{State}\ g$. Just consider the case that f invokes an immediately looping computation, i.e., $f\ l = \bot = \mathsf{State}\ \bot$.[8] The $g = \bot$ here satisfies $P(g)$, but not $\mathit{snd} \circ g = \mathit{id}$.

[8] The equality $\bot = \mathsf{State}\ \bot$ holds by the semantics of **newtype** in Haskell.