Extended Abstract: Reasoning about Effect Parametricity Using Dependent Types

Joris Ceulemans
KU Leuven

Andreas Nuyts
KU Leuven

Dominique Devriese
Vrije Universiteit Brussel

Keywords effect polymorphism, effect parametricity, dependent types

1 Introduction

Consider the following function from the Haskell Prelude.

```
sequence :: Monad m => [m a] -> m [a]
```

It takes a list of monadic computations and performs them sequentially, collecting their results in a list. From this description it is clear that if all monadic computations in a list `ms` of type `[m a]` are pure, then `sequence ms` is pure as well. However, this property does not depend on the implementation of `sequence` but can be shown for every function of the same type. The key observation is that a function of this type must work for any monad and cannot inspect the monad it is instantiated with. Therefore such a function cannot introduce new effects, all it can do is composing and rearranging the effects contained in its argument list. Hence, if this argument list only contains pure monadic computations, the returned result will also be pure.

An argument like the one presented above can be made more precise using effect parametricity, which was done metatheoretically in [4]. In this extended abstract, we describe how properties involving effect parametricity can be proved in ParamDTT, a dependent type system with support for parametricity that was developed in [2].

2 Parametricity

Intuitively, a polymorphic function is called parametric if its implementation does not inspect the type it is instantiated with and therefore applies the same algorithm for every type. In languages like System F or Haskell, every polymorphic function is automatically parametric because the type system does not allow the definition of non-parametric functions. The intuition about parametric polymorphism was formalized by John Reynolds using relations, stating that a parametrically polymorphic function maps related arguments to related results [3]. Using this relational interpretation Philip Wadler subsequently described a procedure to derive from a polymorphic type a theorem which is satisfied by all polymorphic functions of that type [5].

In [4], Janis Voigtländer extended Reynolds's relational interpretation and Wadler's procedure to polymorphism involving type constructor classes such as Monad. From this, he derived properties about functions that quantify parametrically over a monad (so-called effect polymorphic functions) such as the function `sequence` from the introduction.

3 Parametricity and Dependent Types

In Martin-Löf type theory (MLTT), one can use universe types to define polymorphic functions. The polymorphic identity function can for instance be written as

$$\lambda(X : \mathcal{U}) . \lambda(x : X) . x : \Pi(X : \mathcal{U}).X \to X.$$

However Nuyts, Vezzosi and Devriese [2] showed that MLTT does not in general enforce that polymorphic functions defined in this way are parametric; in fact it is possible to define non-parametric polymorphic functions in MLTT without additional axioms. In order to restore parametricity they developed in the same paper ParamDTT, a dependent type system based on MLTT in which the type of a function can contain a parametricity annotation and which provides tools to prove results à la Wadler internally.

More concretely, in ParamDTT we can distinguish between parametric dependent products $\Pi^\sharp(x : A).B$ containing dependent functions that – intuitively – can use the argument x only in type annotations[1], and continuous dependent products $\Pi(x : A).B$ containing functions that can use the argument x in their implementation. Similarly there are parametric dependent sums $\Sigma^\sharp(x : A).B$ whose pairs make their first component available only for parametric use, and continuous dependent sums $\Sigma(x : A).B$ behaving normally.

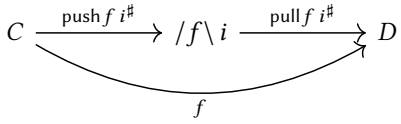
In order to prove parametricity results internally in the type system, ParamDTT provides a way to express relations in the type system. There is an interval type $\mathbb{I}$ with two axiomatic elements $0, 1 : \mathbb{I}$. A relation between two types $C, D : \mathcal{U}$ is then expressed using a continuous function $B : \mathbb{I} \to \mathcal{U}$ with $B\,0 \equiv C$ and $B\,1 \equiv D$, which is called a bridge between C and D. Subsequently, a proof that $c : C$ and $d : D$ are related is expressed using a parametric function $p : \Pi^\sharp(i : \mathbb{I}).B\,i$ with $p\,0^\sharp \equiv c$ and $p\,1^\sharp \equiv d$, which is called a path over B between c and d. The continuous (no symbol) and parametric ($\sharp$) modalities describe the behavior of functions with respect to bridges and paths: continuous functions respect bridges and paths and parametric functions respect paths and strengthen bridges to paths. In fact there is

[1] This intuition is quite appropriate for arguments of type $\mathcal{U}$. However, parametric functions taking an argument of e.g. type $\mathcal{U} \uplus \mathcal{U}$, may still distinguish between types coming from the left and from the right.

TyDe 2019, August 2019, Berlin, Germany
2019.

also a pointwise modality (¶) whose functions respect paths but have no action on bridges.

Additionally, ParamDTT also provides tools to interpret a function $f : C \to D$ as a relation. More concretely, for such a function f there will be a bridge $/f\backslash : \mathbb{I} \to \mathcal{U}$ with $/f\backslash\, 0 \equiv C$ and $/f\backslash\, 1 \equiv D$. Furthermore there are functions $\mathsf{push}\, f : \Pi^\sharp(i : \mathbb{I}).C \to /f\backslash\, i$ and $\mathsf{pull}\, f : \Pi^\sharp(i : \mathbb{I})./f\backslash\, i \to D$ with $\mathsf{push}\, f\, 0^\sharp \equiv \mathsf{id}_C$, $\mathsf{push}\, f\, 1^\sharp \equiv f$, $\mathsf{pull}\, f\, 0^\sharp \equiv f$ and $\mathsf{pull}\, f\, 1^\sharp \equiv \mathsf{id}_D$ These functions allow us to construct a path between any $c : C$ and its image $f\, c : D$.



Finally, there is a path degeneracy axiom asserting that the endpoints of any homogeneous path $p : \Pi^\sharp(i : \mathbb{I}).A$ (i.e. a path over a constant bridge) are equal. More precisely, it gives us an element $\mathsf{path\text{-}to\text{-}eq}\, p : p\, 0 =_A p\, 1$ of the identity type between $p\, 0$ and $p\, 1$ and this is the final piece that allows us to prove parametricity results.

4 Effect Parametricity and Dependent Types

In our current work we are studying how the machinery provided by ParamDTT can be used to prove properties involving effect parametricity, such as the one described in the introduction. For this purpose we define the type of premonads (non-standard terminology) in ParamDTT.

$$
\begin{aligned}
&\mathsf{Premonad} :\equiv \\
&\quad \Sigma(M : \mathcal{U} \to \mathcal{U}). \\
&\quad \Sigma^{¶}(\mathsf{return} : \Pi^\sharp(X : \mathcal{U}).X \to M\,X). \\
&\quad \Sigma^{¶}(\mathsf{bind} : \Pi^\sharp(X, Y : \mathcal{U}).M\,X \to (X \to M\,Y) \to M\,Y).\top
\end{aligned}
$$

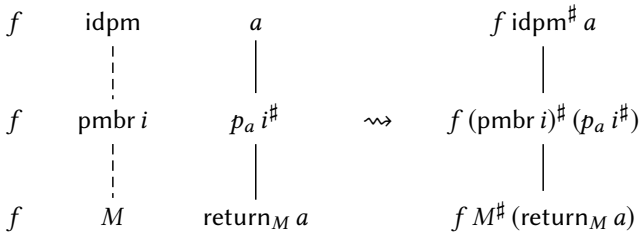
As we can see, a premonad is defined to be a 4-tuple consisting of a type operator, return and bind operations and an element of the unit type $\top$. This last piece of irrelevant information is included because we want a premonad to depend pointwise on its return and bind operations (for technical reasons), which can be done using the $\Sigma^{¶}$-type. The reason we call this a premonad and not a monad is that using dependent types we can express the monad laws and a monad will be a premonad satisfying those laws.[2]

As an example, we will show for any effect polymorphic function $f : \Pi^\sharp(M : \mathsf{Premonad}).M\,A \to M\,A$ (with A an arbitrary closed type) that if M is a monad and $m : M\,A$ is pure, then so is $f\, M^\sharp\, m : M\,A$. This is a statement similar to Theorem 1 in [4]. More precisely, we will define the pure computations in $M\,A$ to be the elements of the form $\mathsf{return}_M\, a$ for some $a : A$ and we will then construct for any $a : A$ a term $\mathsf{thm}_a : \mathsf{return}_M(f\ \mathsf{idpm}^\sharp\ a) =_{M\,A} f\, M^\sharp\, (\mathsf{return}_M a)$ where idpm is the identity premonad.

For this purpose, we use the function $\mathsf{return}_M : A \to M\,A$ to construct a bridge $/\mathsf{return}_M\backslash$ between A and $M\,A$ and a path

$$p_a\, i^\sharp :\equiv \mathsf{push}\, \mathsf{return}_M\, i^\sharp\, a : /\mathsf{return}_M\backslash\, i$$

over this bridge between a and $\mathsf{return}_M\, a$. It is also possible, but a bit more technically involved, to construct a bridge $\mathsf{pmbr} : \mathbb{I} \to \mathsf{Premonad}$ inside the type Premonad between the identity premonad (idpm) and M. In summary, using the bridge and path we have just constructed we get for every $i : \mathbb{I}$ a premonad $\mathsf{pmbr}\, i$ and a corresponding monadic value $p_a\, i^\sharp$. Hence we can apply f to these two arguments as in the following diagram, in which the middle row depends on i reduces to the upper row if $i \equiv 0$ and to the lower if $i \equiv 1$.



Here, the first argument comes from a bridge (dashed line) but as f is parametric in its first argument, this bridge is strengthened to a path. Subsequently, the second argument comes from a path which is respected by f. As a result we have a path

$$f(\mathsf{pmbr}\, i)^\sharp(p_a\, i^\sharp) : /\mathsf{return}_M\backslash\, i$$

between $f\ \mathsf{idpm}^\sharp\ a$ and $f\, M^\sharp\, (\mathsf{return}_M\, a)$. We can then apply $\mathsf{pull}\, \mathsf{return}_M$ to get a path

$$\mathsf{pull}\, \mathsf{return}_M\, i^\sharp\ \left(f(\mathsf{pmbr}\, i)^\sharp(p_a\, i^\sharp)\right) : M\,A$$

between $\mathsf{return}_M(f\ \mathsf{idpm}^\sharp\ a)$ and $f\, M^\sharp\, (\mathsf{return}_M\, a)$. Since this path is homogeneous, the path degeneracy axiom gives us an element of the identity type between its endpoints, which is what we wanted to construct.

In summary, we have seen that free theorems involving effect parametricity can be proved in the dependent type system ParamDTT. The above example is also formalized using the parametric branch of Agda implementing ParamDTT.[3] Formalizations of the theorems that Voigtländer proved metatheoretically [4], are provided there as well.

For more information about the work described in this abstract, we refer to Joris Ceulemans's master's thesis [1].

In the future we intend to investigate how effect parametricity as described in this extended abstract can be applied for practical reasoning about effectful programs in dependently-typed languages.

[2] In fact, what we call a premonad corresponds exactly to an instance of the `Monad` typeclass in Haskell since Haskell cannot impose the monad laws.

[3] See https://github.com/JorisCeulemans/effect-param-agda

Acknowledgments

Andreas Nuyts holds a Ph.D. Fellowship from the Research Foundation - Flanders (FWO).

References

[1] Joris Ceulemans. 2019. *Reasoning about Effect Parametricity Using Dependent Types*. Master of Science in Mathematics. KU Leuven, Belgium. https://github.com/JorisCeulemans/effect-param-agda/blob/master/ThesisJorisCeulemans.pdf

[2] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP, Article 32 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110276

[3] John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*. North Holland, 513–523.

[4] Janis Voigtländer. 2009. Free Theorems Involving Type Constructor Classes: Functional Pearl. In *International Conference on Functional Programming*. ACM, 173–184.

[5] Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture*. ACM, 347–359. https://doi.org/10.1145/99370.99404