Seminar Paper – Theorems for Free!

Kevin Kappelmann

Chair for Logic and Verification

June 22, 2021

Abstract

In his seminal paper, "Theorems for Free!", Wadler [1989] showed the world a nifty trick: by just looking at the type of a polymorphic function, we can derive a theorem that it satisfies. We give a modern recount of this trick, derive some instructive applications, and provide a brief overview of notable extensions of Wadler's trick.

Contents

1 Introduction: Type Systems and Polymorphism

Type systems are lightweight mechanisms that guarantee the absence of certain programming errors such as nontermination, memory leaks, and corruption of data. In their simplest form, they provide guarantees of the shape of values dealt with by an expression. For example, given the following Haskell expression

```
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

a call to `appTwice (+1) "bogus"` will cause a compile-time error while similar expressions in an untyped language often lead to unexpected results or even runtime crashes.

A fundamental problem caused by a type theory is that meaningful programs may not be typeable. For example, one might wish to generalise the definition of `appTwice` such that it works for arbitrary types, not just integers. That is, calling `appTwice (+1) 0` should work just as fine as calling `appTwice (++"l") "haske"`. Note that this is a non-problem in untyped languages in which a definition of `appTwice` accepts any pair of arguments.

A simple solution would be to provide a copy of `appTwice` for any type used in our program, e.g.

```
appTwiceInt :: (Int -> Int) -> Int -> Int
appTwiceInt f x = f (f x)

appTwiceString :: (String -> String) -> String -> String
appTwiceString f x = f (f x)
```

This obviously is not a perfect solution: whenever we want to apply our function to another type of arguments, we need to write a new function for this type, violating the *abstraction principle* of software engineering:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.
>
> — Pierce [2002]

In the example above, the only varying part are the functions' type signatures. So according to the abstraction principle, we should aim to abstract away the concrete types of the signatures and combine them into one common signature. Thankfully, this job has already been done for us. The solution is known as *parametric polymorphism*[1].

Just like functions allow us to abstract over terms by using term variables, parametric polymorphism allows us to abstract over types by using type variables. In Haskell, the parametric version of `appTwice` can be defined as

```
appTwice :: (a -> a) -> a -> a
appTwice f x = f (f x)
```

[1] Parametric polymorphism stands in contrast to *ad-hoc polymorphism*. In this paper, we are only concerned with the former and will simply refer to it as *polymorphism*.

where the type variable `a` is implicitly universally quantified. Parametric polymorphism allows us to respect the abstraction principle while still being protected from unwanted calls such as `appTwice (+1) "bogus"`. But this is not the end of the story; parametric polymorphism comes with another twist:

Polymorphic functions are defined once and for all for any type and as such must work uniformly on values of any type: if we know that `appTwice (+1) 0` applies `(+1)` twice to `0`, then we also know that `appTwice (++"l") "haske"` applies `(++"l")` twice to `"haske"`.

Indeed, as shown by Reynolds [1983] and Wadler [1989], from the type of a polymorphic function, we can derive a theorem that it satisfies. Moreover, every function of the same type satisfies the same theorem. Polymorphic functions provide as *theorems for free*. In this paper, we show you how to prove that such theorems exist and how to get your hands on them, all for free.

2 Historic Context

"Theorems for Free" is an expression and concept introduced by Wadler [1989]. His work is largely based on Reynold's abstraction theorem for the polymorphic lambda calculus. Reynolds [1983] showed how terms evaluated in related environments yield related values for both the simply-typed as well as the polymorphic lambda calculus. The latter proof, however, rested on the wrong conjecture of a set-theoretic model for the polymorphic lambda calculus, later disproved by Reynolds himself [Reynolds, 1984].

Wadler updated Reynolds's abstraction theorem and fixed the proof for the polymorphic case. Moreover, he took Reynold's insights, which were primarily concerned about representation independence and properties of models of the polymorphic lambda calculus, and reformulated them to systematically derive theorems about parametric functions.

Following Wadler's initiative, a variety of extensions for stronger calculi and type systems were proven, commonly referred to as *parametricity theorems*. We shall not be concerned with those extensions as part of this paper but will provide an overview for further reading in Section 5 for the interested reader.

3 Technical Development

In this section, we first introduce the polymorphic lambda calculus and then give an instructive account of how to set up the logical relations needed to prove the Parametricity Theorem. We assume some basic experience with lambda calculi and hence keep the introduction brief. We do not directly follow the development of Wadler [1989] but a more modern recount by Skorstengaard [2019] based on logical relations.

3.1 The Polymorphic Lambda Calculus

The polymorphic lambda calculus – also known as System F – is a typed model of computation introduced by Girard [1972] and Reynolds [1974]. It is an extension of the simply typed lambda calculus that allows for universal quantification over types.

Let us use $\alpha, \alpha_1, \ldots$ to denote type variables and $\tau, \tau_1, \ldots$ to denote types. System F's type syntax can then be defined as follows:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\ \tau$$

In principle, the calculus can be extended with base types like booleans and natural numbers, but as is well known, they can also be encoded using just arrow and universal types (cf Section 4.1).

We use $x, x_1, \ldots$ to denote term variables and $t, t_1, \ldots$ to denote terms. The term (or expression) syntax is then defined as

$$t ::= x \mid \lambda x : \tau.\ t \mid t\, t \mid \Lambda \alpha.\ t \mid t\,[\tau]$$

and the subset of values, denoted by Val, as

$$v ::= x \mid \lambda x : \tau.\ t \mid \Lambda \alpha.\ t$$

In all that follows, we consider terms and types that differ only in the names of bound variables as equal, i.e. they are compared modulo α-equivalence. A term with free variables is called *open* and otherwise *closed*. The grammar of contexts is now defined as follows:

$$\Gamma ::= \bullet \mid \Gamma, x : \tau \mid \alpha$$

We assume that no variable is defined twice in Γ and write $\Gamma(x) = \tau$ if $x : \tau$ is declared in Γ and $\alpha \in \Gamma$ if α is declared in Γ. The (call-by-name) evaluation rules are defined as follows:

$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2}\ \text{E-App1} \qquad \frac{}{(\lambda x : \tau.\ t_1)\, t_2 \to t_1[t_2/x]}\ \text{E-AppAbs}$$

$$\frac{t \to t'}{t\,[\tau] \to t'\,[\tau]}\ \text{E-TApp1} \qquad \frac{}{(\Lambda \alpha.\ t)\,[\tau] \to t[\tau/\alpha]}\ \text{E-TAppAbs}$$

We write $\to^*$ for the reflexive, transitive closure of $\to$ and $=^*$ for the symmetric closure of $\to^*$. Given a term t, we write $t{\downarrow}$ to denote the normal form of t with respect to $\to$. Note that this normal form is unique and always exists since System F is strongly normalising, as originally shown by Girard (see, for example, [Girard et al., 1989]). The evaluation rules are of no great importance for this work but added for the sake of completeness: any other sensible reduction strategy would work just as fine. The well-formed check on types is defined as follows:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{WF-Var} \qquad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\ \tau}\ \text{WF-}\forall \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}\ \text{WF-}\to$$

Finally, the typing rules are given as follows:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2 \quad \Gamma \vdash \tau_1}{\Gamma \vdash \lambda x : \tau_1.\ t : \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\, t_2 : \tau}\ \text{T-App} \qquad \frac{\Gamma, \alpha \vdash t : \tau}{\Gamma \vdash \Lambda \alpha.\ t : \forall \alpha.\ \tau}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t : \forall \alpha.\ \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash t\,[\tau'] : \tau[\tau'/\alpha]}\ \text{T-TApp}$$

We will write $\mathsf{wt}_\tau(t_1, \ldots, t_n)$ to express that all t_i are closed and well-typed with type τ, that is $\bullet \vdash t_i : \tau$ for $1 \le i \le n$. Moreover, we write $t[t_1/x_1, \ldots, t_n/x_n]$ for the term where all occurrences of x_i in t are replaced by t_i (and similarly for type substitutions) and $\rho[t/x]$ for the substitution defined as $\rho[t/x](x) = t$ and $\rho[t/x](y) = \rho(y)$ for $x \neq y$.

3.2 Types as Relations

It is natural to interpret a type τ as a set $[\![\tau]\!]$ containing all values of type τ, e.g. $[\![\texttt{Bool}]\!] = \{\texttt{True}, \texttt{False}\}$ in Haskell[2]. In that interpretation, a judgement $t : \tau$ simply translates to set membership $t \in [\![\tau]\!]$. The key idea to extract free theorems is to give up on that idea and to instead interpret a type as a relation that relates terms of the given type and whose membership is preserved under eliminating forms. The very slogan will be "related terms lead to related results". Here are some instructing examples:

- Base types are interpreted as their identity relation, for example $[\![\texttt{Bool}]\!] = \{(\texttt{True}, \texttt{True}), (\texttt{False}, \texttt{False})\}$.
- Two pairs are related if their components are related, that is $\big((t_1, t_2), (t_1', t_2')\big) \in [\![(\tau_1, \tau_2)]\!] \iff (t_1, t_1') \in [\![\tau_1]\!] \wedge (t_2, t_2') \in [\![\tau_2]\!]$.
- Two lists are related if they have the same length and their elements are related, i.e. $\big([t_1, \ldots, t_n], [t_1', \ldots, t_{n'}']\big) \in [\![\mathsf{List}\ \tau]\!] \iff n = n' \wedge \forall i \in \{1, \ldots, n\}.\ (t_i, t_i') \in [\![\tau]\!]$.
- Two functions are related if they map related arguments to related results, that is $(f, f') \in [\![\tau_1 \to \tau_2]\!] \iff \forall (t, t') \in [\![\tau_1]\!].\ \big(f\, t, f'\, t'\big) \in [\![\tau_2]\!]$.

Some of these examples will not just be true by definition but rather be a consequence of the setup of our logical relation in the next section. We will derive some of them in Section 4.1.

3.3 Logical Relations and the Parametricity Theorem

In order to give a precise definition of our notion of relatedness for System-F, we will make us of a technique called *logical relations*. The term *logical relation* originates from Plotkin [1973] but similar ideas can be traced back to Tait [1967] and his proof of the strong normalisation of the simply typed lambda calculus. Logical relations proved to be a strong tool in programming language theory. For example, they can be used to prove strong normalisation, type safety, the equivalence of programs, and, to our great interest, free theorems for various lambda calculi.

Broadly speaking, a logical relation $R[\![\tau]\!]$ is an inductive family of relations indexed by types. In general, a logical relation can be constructed with respect to a suitable interpretation (model) of the calculus in question. In our case, we can stick to the simple case where we interpret terms as their syntactic representation. If we then want to prove that the terms of our language satisfy some n-ary property P, we usually construct $R[\![\tau]\!]$ in a way such that:

1. If $R[\![\tau]\!]\big(t_1, \ldots, t_n\big)$ then
   (a) $\mathsf{wt}_\tau(t_1, \ldots, t_n)$
   (b) $P\big(t_1, \ldots, t_n\big)$
2. The conditions of the relation are preserved by eliminating forms.

It then suffices to prove that all well-typed terms of our language satisfy the logical relation to obtain our proof of property P. Needless to say, this is just a vague recipe that needs further refinement depending on the property P at hand.

[2] ignoring that `undefined :: Bool` for a moment

In our case, we want to prove a property of relatedness as exemplified in the previous section. Indeed, our goal will be to show that all terms t are related to themselves. Now that sounds rather dull at first glance, but the magic kicks in once we think about how to relate polymorphic terms – the reader will have to grant us a credit of trust at this point.

For ease of exposition, we split our logical relation into two parts:

1. a logical relation $\mathcal{V}[\![\tau]\!]$ on values that encodes the notion of relatedness that we want to preserve, and
2. a logical relation $\mathcal{E}[\![\tau]\!]$ on general terms that reduces the relatedness property to the simpler case of values.

Pushing aside the existence of type variables for a moment, let us begin with the simple case of function types: As mentioned in the previous section, two functions should be related if they map related arguments to related results. Following our recipe, a first approximate definition could thus look as follows:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] \coloneqq \Big\{(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \mid \mathsf{wt}_{\tau_1 \to \tau_2}(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \wedge \\ \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!].\ \big(t_1[v_1/x], t_2[v_2/x]\big) \in \mathcal{E}[\![\tau_2]\!]\Big\}$$

In words: two lambda-abstractions are related if they are well-typed on the same type and the relatedness property is maintained on the eliminating form of the terms, that is function application. Since the terms $t_1[v_1/x]$ and $t_2[v_2/x]$ are not necessarily values, we have to push the check of the latter to $\mathcal{E}[\![\tau_2]\!]$.

Similarly, we have to deal with type abstractions. Again, we can take our recipe and to get a first, approximate definition:

$$\mathcal{V}[\![\forall \alpha.\ \tau]\!] \coloneqq \Big\{(\Lambda \alpha.\ t_1, \Lambda \alpha.\ t_2) \mid \mathsf{wt}_{\forall \alpha.\, \tau}(\Lambda \alpha.\ t_1, \Lambda \alpha.\ t_2) \wedge \\ \forall \tau_1, \tau_2.\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau[?/\alpha]]\!]\Big\},$$

where $\bullet \vdash \tau_1$ and $\bullet \vdash \tau_2$ (we will silently make this assumption in all upcoming definitions). But here we face a problem: there is not a single type the terms $t_1[\tau_1/\alpha]$ and $t_2[\tau_2/\alpha]$ can be related under, leaving a puzzling ? in our definition. The idea now is to not substitute any type for α in τ at all; instead, we keep track of the chosen types τ_1, τ_2 in an explicit substitution ρ that will then be used to interpret the free type variable α in τ when needed. The substitution thus becomes an additional parameter of the entire logical relation. We hence update our definition as follows:

$$\mathcal{V}[\![\forall \alpha.\ \tau]\!]_\rho \coloneqq \Big\{(\Lambda \alpha.\ t_1, \Lambda \alpha.\ t_2) \mid \mathsf{wt}_{\forall \alpha.\, \rho(\tau)}(\Lambda \alpha.\ t_1, \Lambda \alpha.\ t_2) \wedge \\ \forall \tau_1, \tau_2.\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha \mapsto (\tau_1, \tau_2)]}\Big\},$$

where $\mathsf{wt}_{\rho(\tau)}(t_1, t_2) \coloneqq \mathsf{wt}_{\rho_1(\tau)}(t_1) \wedge \mathsf{wt}_{\rho_2(\tau)}(t_2)$ with $\rho_i(\alpha) \coloneqq \tau_i$ if $\rho(\alpha) = (\tau_1, \tau_2)$. This leaves us with the definition of the value interpretation for type variables $\mathcal{V}[\![\alpha]\!]_\rho$. Here is a first approximation:

$$\mathcal{V}[\![\alpha]\!]_\rho \coloneqq \big\{(v_1, v_2) \mid \mathsf{wt}_{\rho(\alpha)}(v_1, v_2) \wedge ?\big\}$$

Once again, we run into a problem: we know $\rho(\alpha) = (\tau_1, \tau_2)$ but τ_1 might be different from τ_2. So the question becomes: how should we relate values of possibly different types? Intuitively, it seems like there should not be any strong restriction on the relation of the values: A polymorphic definition is independent of the concrete types being chosen at runtime and hence works uniformly on values of any type. As such, whenever we pick two types τ_1, τ_2 to substitute for α in a polymorphic definition, it seems like we should be able to pick any relation on τ_1 and τ_2 and that relation should be preserved by the polymorphic definition. This idea leads us to our final updated definition for type abstractions:

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho \coloneqq \Big\{(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid \mathsf{wt}_{\forall\alpha.\,\rho(\tau)}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2)\ \wedge \forall\tau_1, \tau_2, R \in \mathcal{R}(\tau_1, \tau_2).\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2,R)]}\Big\},$$

where any R is just a relation of closed, well-typed values, that is

$$\mathcal{R}(\tau_1, \tau_2) \coloneqq \big\{R \in \mathcal{P}(Val \times Val) \mid \forall(v_1, v_2) \in R.\ \mathsf{wt}_{\tau_1}(v_1) \wedge \mathsf{wt}_{\tau_2}(v_2)\big\}.$$

As we will see in Section 4, even though two polymorphic terms should be related for any choice of R, not all choices will give us useful theorems. Indeed, the key to derive meaningful theorems will primarily consist of finding a good choice for R.

We are now able to finish our value interpretation for type variables:

$$\mathcal{V}[\![\alpha]\!]_\rho \coloneqq \big\{(v_1, v_2) \in R \mid \mathsf{wt}_{\rho(\alpha)}(v_1, v_2) \wedge \rho(\alpha) = (\tau_1, \tau_2, R)\big\}$$

Now we can update our definition on function types to account for all of above changes:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho \coloneqq \Big\{\big(\lambda x : \rho(\tau_1).\ t_1, \lambda x : \rho(\tau_1).\ t_2\big) \mid\ \mathsf{wt}_{\rho(\tau_1)\to\rho(\tau_2)}\big(\lambda x : \rho(\tau_1).\ t_1, \lambda x : \rho(\tau_1).\ t_2\big) \wedge \forall(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho.\ \big(t_1[v_1/x], t_2[v_2/x]\big) \in \mathcal{E}[\![\tau_2]\!]_\rho\Big\}$$

Having finished our interpretation of values, we can easily extend it to terms:

$$\mathcal{E}[\![\tau]\!]_\rho \coloneqq \big\{(t_1, t_2) \mid \mathsf{wt}_{\rho(\tau)}(t_1, t_2) \wedge (t_1{\downarrow}, t_2{\downarrow}) \in \mathcal{V}[\![\tau]\!]_\rho\big\}$$

So all that $\mathcal{E}[\![\tau]\!]$ does is normalising the terms and delegating the check to $\mathcal{V}[\![\tau]\!]$. Note that the logical relations $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{E}[\![\cdot]\!]$ are mutually dependent, and hence it is worth to mention that they are well-founded: every definition of $\mathcal{E}[\![\tau]\!]$ reduces to a check in $\mathcal{V}[\![\tau]\!]$, which in turn might rely on a check of $\mathcal{E}[\![\tau']\!]$ for a syntactically smaller type of τ. So the relations are indeed well-founded on the syntactic complexity of types.

Following our recipe, we now wish to show that all well-typed terms of our language satisfy the logical relation. Of course, not all terms of the same type are related to each other, but any term t should at least be related to itself. So what we could prove is that whenever $\mathsf{wt}_\tau(t)$ then $(t, t) \in \mathcal{E}[\![\tau]\!]$. That might seem a bit mundane, but remember that for two related polymorphic terms, the definition of $\mathcal{V}[\![\forall\alpha.\,\tau]\!]$ tells us that we can substitute two different types for α and the terms stay related. Repeatedly substituting different types in a given t will then give us two quite different programs, which are still related.

Now as for the proof, it seems natural to proceed by induction on the typing derivation of t. However note that as part of the derivation, the typing context will

change and we will have to deal with open terms. So we actually need to prove a more general property that deals with contexts and uses them to close off open terms. For this, we give an interpretation of contexts:

$$\begin{aligned}
[\![\bullet]\!] &\coloneqq \{\emptyset\} \\
[\![\Gamma, \alpha]\!] &\coloneqq \{\rho[\alpha \mapsto (\tau_1, \tau_2, R)] \mid \rho \in [\![\Gamma]\!] \wedge \bullet \vdash \tau_1 \wedge \bullet \vdash \tau_2 \wedge R \in \mathcal{R}(\tau_1, \tau_2)\} \\
[\![\Gamma, x : \tau]\!] &\coloneqq \{\rho[x \mapsto (v_1, v_2)] \mid \rho \in [\![\Gamma]\!] \wedge (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\rho\}
\end{aligned}$$

We can now define our wanted notion of relatedness

$$(\Gamma \vdash t_1 \approx t_2 : \tau) \coloneqq \forall \rho \in [\![\Gamma]\!].\ \big(\rho_1(t_1), \rho_2(t_2)\big) \in \mathcal{E}[\![\tau]\!]_\rho$$

and finally state the main theorem:

Theorem 1 (Parametricity Theorem)**.** *If* $\Gamma \vdash t : \tau$ *then* $\Gamma \vdash t \approx t : \tau$.

The proof of the theorem is by induction on the typing derivation of t. We omit the details – they can be found, for example, in [Skorstengaard, 2019].

We will simply write $t_1 \approx t_2$ for $\bullet \vdash t_1 \approx t_2 : \tau$ if τ is clear from the context. Theorems that follow by instantiating the Parametricity Theorem are called "free theorems". Let us explore some applications of the theorem in the next section.

4 Examples of Free Theorems

4.1 Booleans, Pairs, and Natural Numbers

In Section 3.2 we gave some instructive examples of the notion of relatedness for basic inductive types such as booleans as well as pairs and lists. Our calculus does not include any of these types as primitives, but it is simple to construct them using our existing tools. We will call such encodings *functional*.

Let us begin with the simple case of booleans: we can define its type as $\mathsf{Bool} \coloneqq \forall\alpha.\, \alpha \to \alpha \to \alpha$ and its two constructors $\mathsf{true} \coloneqq \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x$ and $\mathsf{false} \coloneqq \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y$.

Example 2 ($\mathsf{true} \not\approx \mathsf{false}$)**.** Let us show that $\mathsf{true} \not\approx \mathsf{false}$. Pick $\tau_1 = \tau_2 = \mathsf{Bool}, v_1 = \mathsf{true}, v_2 = \mathsf{false}$ and $R = \{(v_1, v_1), (v_2, v_2)\}$. Then $(v_1, v_1), (v_2, v_2) \in R, \mathsf{true}\,[\tau_1]\,v_1\,v_2 \to^* v_1, \mathsf{false}\,[\tau_2]\,v_1\,v_2 \to^* v_2$ but $(v_1, v_2) \notin R$. Hence $\big((\mathsf{true}\,[\tau_1]\,v_1\,v_2)\downarrow, (\mathsf{false}\,[\tau_2]\,v_1\,v_2)\downarrow\big) \notin R$ and $\mathsf{true} \not\approx \mathsf{false}$.

The Parametricity Theorem is often used to show that two terms t_1 and t_2 "behave the same", meaning that the terms are indistinguishable when replaced by each other in any program context. The terms t_1 and t_2 are then said to be *observationally equivalent*.

Example 3 (terms of type Bool)**.** Let us explore how many observationally different closed terms of type Bool there are. For this, assume we are given an arbitrary value $t : \mathsf{Bool}$ (we could also pick an arbitrary term and normalise it first). We want to explore the behaviour of t, so assume we are given some arbitrary τ and $v_1, v_2 : \tau$. What are the possible results of $t\,[\tau]\,v_1\,v_2$?

By the Parametricity Theorem, we have $t \approx t$. So we can pick $\tau' = \mathsf{Bool}, v_1' = \mathsf{true}$, $v_2' = \mathsf{false}, R = \{(v_1', v_1), (v_2', v_2)\}$ and obtain $\big(t\,[\tau']\,v_1'\,v_2', t\,[\tau]\,v_1\,v_2\big) \in \mathcal{E}[\![\alpha]\!]_{[\alpha \mapsto (\tau', \tau, R)]}$.

So there is $v' : \tau'$ and $v : \tau$ with $t\,[\tau']\,v_1'\,v_2' \to^* v'$ and $t\,[\tau]\,v_1\,v_2 \to^* v$ and $(v', v) \in R$. Thus either $v' = v_1' = \mathsf{true}$ or $v' = v_2' = \mathsf{false}$. In case of the former, we get $v = v_1$ and thus $t\,[\tau]\,v_1\,v_2 \to^* v_1$. In case of the latter, we get $v = v_2$ and thus $t\,[\tau]\,v_1\,v_2 \to^* v_2$. Now τ, v_1, v_2 were chosen arbitrarily and $t\,[\tau']\,v_1'\,v_2' \to^* v'$ holds independently of that choice. Hence $\forall \tau, v_1, v_2.\, t\,[\tau]\,v_1\,v_2 \to^* v_1$ or $\forall \tau, v_1, v_2.\, t\,[\tau]\,v_1\,v_2 \to^* v_2$; in other words, t is observationally equivalent to either true or false.

Example 4 (pairs)**.** In a similar spirit, we can define the type of pairs for fixed types τ_1, τ_2 as $(\tau_1, \tau_2) \coloneqq \forall \alpha.\, (\tau_1 \to \tau_2 \to \alpha) \to \alpha$ and the constructor

$$\mathsf{pair} \coloneqq \Lambda \alpha_1.\, \Lambda \alpha_2.\, \lambda x_1 : \alpha_1.\, \lambda x_2 : \alpha_2.\, \Lambda \alpha_3.\, \lambda f : \alpha_1 \to \alpha_2 \to \alpha_3.\, f\, x_1\, x_2.$$

Note that $\mathsf{pair}\,[\tau_1]\,[\tau_2]\,v_1\,v_2 : (\tau_1, \tau_2)$ for any $v_1 : \tau_1$ and $v_2 : \tau_2$. Now by the Parametricity Theorem, for any $\tau_1, \tau_1', \tau_2, \tau_2', v_1, v_1', v_2, v_2', R_1, R_2$, we have

$$\big(\mathsf{pair}\,[\tau_1]\,[\tau_2]\,v_1\,v_2, \mathsf{pair}\,[\tau_1']\,[\tau_2']\,v_1'\,v_2'\big) \in \mathcal{E}[\![(\alpha_1, \alpha_2)]\!]_{[\alpha_1 \mapsto (\tau_1, \tau_1', R_1), \alpha \mapsto (\tau_2, \tau_2', R_2)]}$$

whenever $(v_1, v_1') \in R_1$ and $(v_2, v_2') \in R_2$. In other words: pairs are related whenever their components are related.

Example 5 (natural numbers)**.** We can encode natural numbers in our system using church numerals. The type of natural numbers can be defined as $\mathsf{Nat} \coloneqq \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$, the number zero as $\mathsf{0} \coloneqq \Lambda \alpha.\, \lambda s : \alpha \to \alpha.\, \lambda z : \alpha.\, z$ and the successor function as $\mathsf{succ} \coloneqq \lambda n : \mathsf{Nat}.\, \Lambda \alpha.\, \lambda s : \alpha \to \alpha.\, \lambda z : \alpha.\, s\ (n\,[\alpha]\,s\,z)$. Let us define $f^0\,x \coloneqq x$ and $f^{n+1} \coloneqq f\,(f^n x)$. The n-th Church numeral can then be defined by $\mathsf{n} \coloneqq \mathsf{succ}^n\,0$. Note that the type of `appTwice` from our introduction actually has type Nat! Indeed, in a similar style to Example 3, we can derive a free theorem for any term of that type:

Assume we are given some value $t : \mathsf{Nat}$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. Pick $\tau' = \mathsf{Nat}, s' = \mathsf{succ}, z' = \mathsf{0}$, and $R = \big\{(\mathsf{n}{\downarrow}, (s^n\,z){\downarrow}) \mid n \in \mathbb{N}\big\}$. First note that $(\mathsf{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\tau', \tau, R)]}$: If $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{[\alpha \mapsto (\tau', \tau, R)]}$ then $(v_1, v_2) = \big(\mathsf{n}{\downarrow}, (s^n\,z){\downarrow}\big)$ for some n. Thus $(\mathsf{succ}\,v_1, s\,v_2) =^* \big((\mathsf{n}+1){\downarrow}, (s^{n+1}\,z){\downarrow}\big) \in R$.

Hence, we get $\big(t\,[\mathsf{Nat}]\,\mathsf{succ}\,\mathsf{0}, t\,[\tau]\,s\,z\big) \in \mathcal{E}[\![\alpha]\!]_{[\alpha \mapsto (\tau', \tau, R)]}$ by the Parametricity Theorem. So there is $v' : \mathsf{Nat}$ and $v : \tau$ such that $t\,[\tau']\,\mathsf{succ}\,\mathsf{0} \to^* v'$ and $t\,[\tau]\,s\,z \to^* v$ and $(v', v) \in R$. Hence $v =^* s^n\,z$ where n is determined by $t\,[\mathsf{Nat}]\,\mathsf{succ}\,\mathsf{0} \to^* v' =^* \mathsf{n}$. Putting it all together, we discovered that t is observationally equivalent to the n-th Church numeral – in the case of `appTwice`, the Church numeral 2.

Instead of relating the functional encoding of natural numbers with terms of type Nat, we could also add the natural numbers as a primitive to our calculus and then, by the same process, show that the primitive natural numbers are in an isomorphic relation to their functional encodings of type Nat. This result indeed generalises to all functional encodings of inductive types as shown by Plotkin and Abadi [1993].

4.2 Lists

Just as in the previous section, we could encode the type of lists for a fixed type τ in our calculus by using the type $\mathsf{List}\ \tau \coloneqq \forall \alpha.\, (\alpha \to \tau \to \tau) \to \tau \to \tau$ defining suitable nil and cons constructors, defining the well-known map function, etc. As mentioned in Section 3.2, it would then turn out that two lists are related if they have the same length and their elements are related. More formally:

$$\big([v_1, \ldots, v_n], [v_1', \ldots, v_{n'}']\big) \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_\rho \iff n = n' \wedge \forall i \in \{1, \ldots, n\}.\, (v_i, v_i') \in \mathcal{V}[\![\alpha]\!]_\rho$$

However, for the sake of brevity, we will not explicitly derive this but simply assume that we are given suitable definitions for the terms of interest. The following examples are taken from Wadler [1989].

Example 6 (rearranging lists)**.** Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$. Intuitively, t can do nothing but re-arrange, copy, and delete elements from a given input list. In particular, we would expect the following theorem to hold:

$$\forall\alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\ \alpha).\, \mathsf{map}\, f\, (t\,[\alpha]\, xs) =^* t\,[\alpha']\, (\mathsf{map}\, f\, xs)$$

Well lucky us: this is indeed a free theorem! Pick any $\tau, \tau', f : \tau \to \tau'$. By the Parametricity Theorem, we have $t \approx t$. So for any R and $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$, we have $(t\,[\tau]\, xs, t\,[\tau']\, xs') \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$. Note that if we specialise $R = \{(v, (f\, v)\!\downarrow) \mid \mathsf{wt}_\tau(v)\}$ to be the extensional representation of $f\,[\tau]\,[\tau']$, the property $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ simply translates to $xs' =^* \mathsf{map}\, f\, xs$. And similarly, the property $(t\,[\tau]\, xs, t\,[\tau']\, xs') \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ translates to $t\,[\tau']\, xs' =^* \mathsf{map}\, f\, (t\,[\tau]\, xs)$. Putting it together, we get $t\,[\tau']\, (\mathsf{map}\, f\, xs) =^* \mathsf{map}\, f\, (t\,[\tau]\, xs)$, which is what we wanted to show.

Example 7 (sorting lists)**.** Assume we have a value $s : \forall\alpha.\, (\alpha \to \alpha \to \mathsf{Bool}) \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$ (for example, Haskell's `sortBy` and `nubBy` functions). By the Parametricity Theorem, we can pick $\tau, \tau', R, \mathsf{cmp}, \mathsf{cmp}'$ such that $(s\,[\tau]\, \mathsf{cmp}, s\,[\tau']\, \mathsf{cmp}') \in \mathcal{E}[\![\mathsf{List}\ \alpha \to \mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau', \tau', R)]}$. Now similarly to the previous example, pick an arbitrary $f : \tau \to \tau'$ and let $R = \{(v, (f\, v)\!\downarrow) \mid \mathsf{wt}_\tau(v)\}$ be the representation of f. Then, for any list xs, we get $\mathsf{map}\, f\, (s\,[\tau]\, \mathsf{cmp}\, xs) =^* s\,[\tau']\, \mathsf{cmp}'\, (\mathsf{map}\, f\, xs)$ whenever $(\mathsf{cmp}, \mathsf{cmp}') \in \mathcal{V}[\![\alpha \to \alpha \to \mathsf{Bool}]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$, which in turn unfolds to $\mathsf{cmp}\, t_1\, t_2 =^* \mathsf{cmp}'\, (f\, t_1)\, (f\, t_2)$ for all $t_1 : \tau, t_2 : \tau$. In other words: map commutes with s if the mapped function is order-preserving.

4.3 Indefinability of `undefined` and Parametric Equality

Free theorems do not necessarily need to have a positive character but can also provide useful negative results:

Example 8. Note that in Haskell `undefined :: forall a. a`. Can we define such a term in System F? Assume there is a value $u : \forall\alpha.\, \alpha$. Pick any τ, τ' and set $R = \emptyset$. Then by the Parametricity Theorem, $(u\,[\tau], u[\tau']) \in R = \emptyset$, which is impossible.

The following insight is taken from Wadler [1989].

Example 9. Assume we are given a polymorphic equality function $\mathsf{eq} : \forall\alpha.\, \alpha \to \alpha \to \mathsf{Bool}$. Remember that $\mathcal{V}[\![\mathsf{Bool}]\!] = \{(\mathsf{true}, \mathsf{true}), (\mathsf{false}, \mathsf{false})\}$. Then by the Parametricity Theorem, for any $\tau, \tau', f : \tau \to \tau', v_1 : \tau, v_2 : \tau$, we get $\mathsf{eq}\,[\tau]\, v_1\, v_2 =^* \mathsf{eq}\,[\tau']\, (f\, v_1)\, (f\, v_2)$. However, this should only be the case for injective functions f and not in general. Hence, polymorphic equality is undefinable in the System F.

As mentioned by Wadler [1989], this example motivates the introduction of restricted type quantification. For example, Haskell uses the notion of type classes [Wadler and Blott, 1997] to define its equality function `(==) :: Eq a => a -> a -> Bool`. Thankfully, as already sketched by Wadler [1989] and finally formalised and proven

by Voigtländer [2009], the Parametricity Theorem can be adapted to work with type classes[3] and even type constructors.

5 Free Theorems and Beyond

Following Wadler, many authors have taken up the task of extending the Parametricity Theorem to other type systems. Some notable examples of generalisations and applications of the results presented here can be found in the following:

- As already explained by Reynolds [1983], the Parametricity Theorem can be used to show that two implementations of the same abstract data type are observationally equivalent (this is often called *representation independence*).
- Wadler [1989] showed that the Parametricity Theorem can be adapted to non-total extensions of System F with general recursion, though the theorems derived become weaker: all relations considered must be strict in the sense that $(\bot, \bot) \in R$. In particular, for functional relations, this means that diverging arguments cause diverging results.
- The introduction of general recursive types causes problems in the well-foundedness of the logical relation: For example, if we naively extend our logical relation to cope with the type of boolean lists $\mathsf{List\ Bool} = \mathsf{unit} + \mathsf{Bool} \times \mathsf{List\ Bool}$, we end up with the following definition $\mathcal{V}[\![\mathsf{List\ Bool}]\!]_\rho = \mathcal{V}[\![\mathsf{unit}]\!]_\rho \cup \mathcal{V}[\![\mathsf{Bool} \times \mathsf{List\ Bool}]\!]_\rho$, leading to a circularity. The solution to this issue was given by Appel and McAllester [2001] and nowadays known as *step-indexing*: the logical relation is not just indexed by types anymore but also by the number of reduction steps we allow ourselves to perform on terms and applications of values.
- As mentioned in Section 4.3, Voigtländer [2009] extended the Parametricity Theorem to type classes and type constructors.
- Bernardy et al. [2012] generalised the parametricity results to pure type systems and its extension with inductive types (and hence in particular to dependent type systems).
- Ahmed et al. [2017] showed that gradually typed languages preserve some parametricity results.
- On the more practical side, various tactics for interactive theorem provers have been developed to automatically derive free theorems from given user definitions [Huffman and Kunčar, 2013, Tabareau et al., 2021].
- The ongoing Iris-project [Jung et al., 2018] is a verification framework that provides mechanisms to work with logical relations on higher-order languages with mutable state.

We want to conclude the paper by answering a question posed by Wadler in his seminal paper:

> How useful are the [free] theorems so generated? Only time and experience will tell...

[3] The basic idea is that the logical relation for type variables restricted by some class constraints only ranges over those types that satisfy the constraints and those relations that preserve the constraints.

Nearly 32 years later, I allow myself to reply positively: the idea of free theorems kicked off much fruitful research and their results can indeed be very useful in formal verification.

References

Amal Ahmed, Dustin Jamner, Jeremy G Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

Andrew W. Appel and David McAllester. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001. ISSN 0164-0925. URL https://doi.org/10.1145/504709.504712.

Jean-Philippe Bernardy, P Jannson, and RA Paterson. Proofs for Free – Parametricity for Dependent Types. *Journal of Functional Programming*, 22(2):107–152, 2012.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*, volume 7. Cambridge university press, 1989.

Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A Modular Design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 131–146. Springer, 2013.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

Gordon Plotkin and Martín Abadi. A Logic for Parametric Polymorphism. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 361–375, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47586-6.

Gordon D Plotkin. Lambda-Definability and Logical Relations. *Memorandum SAI-RM*, 4, 1973.

John C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, 1983.

John C. Reynolds. Polymorphism is not Set-Theoretic. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 145–156, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-540-38891-3.

Lau Skorstengaard. An Introduction to Logical Relations. *arXiv preprint arXiv:1907.11133*, 2019.

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The Marriage of Univalence and Parametricity. *Journal of the ACM (JACM)*, 68(1):1–44, 2021.

William W Tait. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

Janis Voigtländer. Free Theorems Involving Type Constructor Classes: Functional Pearl. *ACM Sigplan Notices*, 44(9):173–184, 2009.

Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

Philip Wadler and Stephen Blott. How to Make Ad-Hoc Polymorphism Less Ad Hoc. 08 1997. doi: 10.1145/75277.75283.